Automated Software Engineering (2023) 30:8
https://doi.org/10.1007/s10515-022-00372-8

# Madusa: mobile application demo generation based on usage scenarios

Jaehyung Lee[1] · Hangyeol Cho[1] · Woosuk Lee[1]

Received: 15 March 2022 / Accepted: 20 November 2022 / Published online: 12 January 2023

**Abstract**
Mobile applications have grown rapidly in size. This dramatic increases in size and complexity make mobile applications less accessible to a broader scope of users. The prevailing approach for better accessibility of mobile applications is to manually reimplement slimmed versions with a small but representative portion of a regular original app. Unfortunately, this approach imposes significant burden on developers. We propose a system called Madusa to enable developers to effectively customize and reduce their mobile applications for Android. Madusa takes as input an original app, an upper bound on the size of a reduced version, and usage scenarios as a high-level specification of its desired core functionality. The output is a reduced version of the app that is still correct with respect to the specification while not exceeding the size limit. Madusa constructs a graph representing dependencies among methods and resources and identifies a sub-part of the graph using integer linear programming to generate a reduced version that exhibits behaviors as similar as possible to the original app. Our experimental evaluation on a suite of 19 Android apps available on Google Play Store. Madusa effectively converges to the desired simplified apps by reducing the app size by 40% on average (maximally by 60%). We conclude our approach effectively removes redundant code and resources with respect to given usage scenarios.

**Keywords** Demo generation · Mobile applications · Integer linear programming · Android · Static analysis for android

## 1 Introduction

Mobile applications (apps for short) have grown rapidly in size. The average Android APK (Android application package) size has grown by over five times since 2012 (Henderson et al. 2018). In addition, the top 10 iPhone apps by downloads in

✉ Woosuk Lee
woosuk@hanyang.ac.kr

1 Department of Computer Science & Engineering, Hanyang University, Ansan, Korea

the U.S. for 2021 require 2.2 GB of storage in total, which is four times larger than five years ago. A major part of this growth comes with the continuous addition of new features. As a representative example, due to Gmail's expanding feature set, its mobile app is 18x larger than five years ago, growing from 19 to 355 MB as of 2021 (The iPhone's Top Apps Are Nearly 4x Larger Than Five Years Ago 2021).

The dramatic increases in size and complexity make mobile applications less accessible to a broader scope of users. A large application size demotivates users to install the application because mobile data and device storage are both at a premium. Such a large app is particularly not accessible to low-end devices that have limited computing resources. As a result, it is known that the *application install rate* (the proportion of store visitors who install) is inversely proportional to application size in general (Henderson et al. 2018).

The prevailing approach for better accessibility of mobile applications is to reimplement slimmed versions with a small but representative portion of a regular original app, complying with restrictions over the size of the simplified versions. Representative examples are Android instant app and iOS App Clip. An Android instant app is a small application that enables end users to test out a portion of a native Android app without installing it. Developers can choose to create an instant app from scratch or transform a full-fledged app into an instant app. Turning a traditional native app into an instant app requires developers to modularize the app into separate code components. Instant apps should be smaller than 15 MB to launch quickly. An iOS App Clip is a fast and lightweight small part of an original app. The goal is similar to that of Android instant apps (i.e., enabling testing out a portion of an app without installation). Apple App Clips also require developers to refactor apps' code to be modular and reusable. Uncompressed App Clips should be less than 10 MB to launch instantly.

Unfortunately, refactoring original apps complying with the size restrictions at the same time is quite non-trivial, which is why these features are sparingly used. For example, less than 0.1% of all the Android applications except games provide their instant applications (Android Instant Apps - Android SDK statistics 2021). In addition, out of the 38 applications demonstrated as notable cases of Android instant apps by Google (Apps to Try Now - Android Apps on Google Play 2021; Google Play Instant Developer Success Stories 2021), only 19 instant applications remain available to date. Similarly, only a handful of iOS apps currently provide their lightweight versions for App Clips to date.

We present a useful system to enable users to customize and reduce Android apps.[1] The system takes as input an app to be simplified, an upper bound on the size of the simplified version, and usage scenarios that exercise its desired core functionality. Then the system generates a simplified version of the app that is smaller than the size limit but still able to exercise the given usage scenarios.

Figure 1 depicts a high-level architecture of our system MADUSA. As input, MADUSA takes an original app, an upper bound $n$ of the size of a slimmed version,

[1] Though our proof-of-concept targets Android, the approach is general and agnostic to the underlying OS, thus we believe it will be potentially applicable to iOS as well.

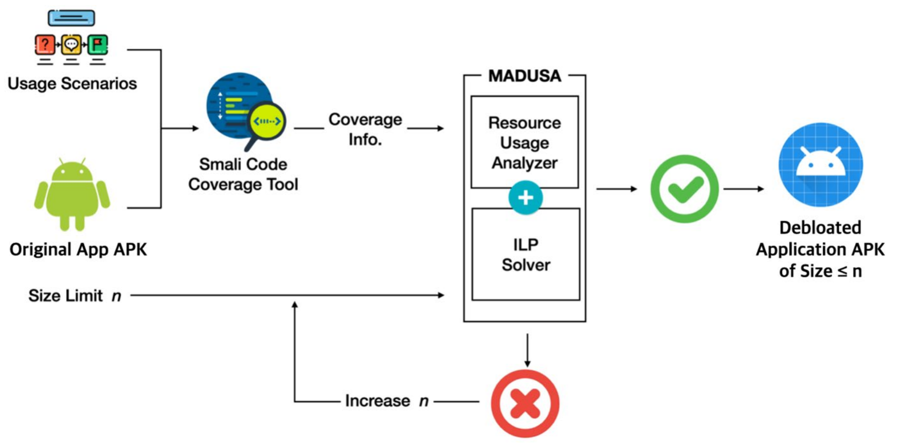


**Fig. 1** Overview of the MADUSA system

and usage scenarios demonstrating the desired functionality. We consider the setting where apps are provided as APK files (i.e., already compiled and packaged), so that MADUSA can be generally applicable even when the entire source code is not available for various reasons (e.g., apps using closed-source libraries, legacy apps). In this setting, while exercising the usage scenarios, MADUSA first measures code coverage by instrumenting Dalvik bytecode in its `smali` (Github - smali 2021) representation.

When with coverage information, an immediate method for generating a desired lightweight version is to package an app with the instructions covered in the usage scenarios. In addition to the code, additional resource files the code uses (e.g., bitmap images, layout definitions) also would have to be included in the resulting app.

However, this simple-minded approach may compromise the *robustness* of the resulting app because the usage scenarios may not be exhaustive enough to demonstrate the desired functionality. For example, the resulting app may crash on events never seen in the usage scenarios.

To improve the robustness, we include as much original instructions as possible in the resulting app as long as the size limit is not exceeded. That is because the more instructions are included in the resulting app, the more similar behaviors the resulting app will exhibit compared to the original app.

We reduce this optimization problem into an integer linear programming (ILP) instance: given instructions along with information about code coverage and resource files the code uses, what are the maximum subset of the instructions that can lead to an app smaller than the size limit? With a solution found by an off-the-shelf ILP solver, MADUSA effectively generates a reduced but robust app which can exercise the desired functionality and comply with the size restriction at the same time. No existing solution means the size constraint is too strict to be satisfied. Then, MADUSA increase the upper bound *n* and repeats the process.

We evaluate MADUSA on a suite of 19 Android apps available on Google Play Store. MADUSA effectively converges to the desired simplified apps. It could reduce the app size by 40% on average, (code reduction 25%, resource reduction 50% respectively). The robustness of the reduced apps is validated by running a state-of-the-art fuzzer Monkey (UI/Application Exerciser Monkey 2021) on the apps. The apps reduced by our system did not crash in 96% of event sequences randomly generated by Monkey.

In summary, the paper makes the following contributions.

- We propose a general method for reducing the size of mobile applications. It aims to remove unwanted functionalities from original mobile applications to improve their accessibility to a broader scope of users.
- We evaluate Madusa using a set of real-world Android apps. Our experiments show that it effectively reduces the size of apps. All the experimental data and our tool are publicly available.[2]

## 2 Motivating example

We illustrate how Madusa enables users to customize and reduce apps using the example of `NOS`, a Dutch news application. Suppose the user wants to obtain a simplified version of `NOS` to generate an Android instant app to allow end users to use a representative portion of the app. We need to trim the original app since the size is 22 MB whereas any instant app should not exceed 15 MB. There exists an official instant app of `NOS`, which was manually written by the original developers. We demonstrate how to automatically obtain a reduced version that has the same functionality as the instant app of `NOS` by using Madusa.

### 2.1 Inputs to Madusa

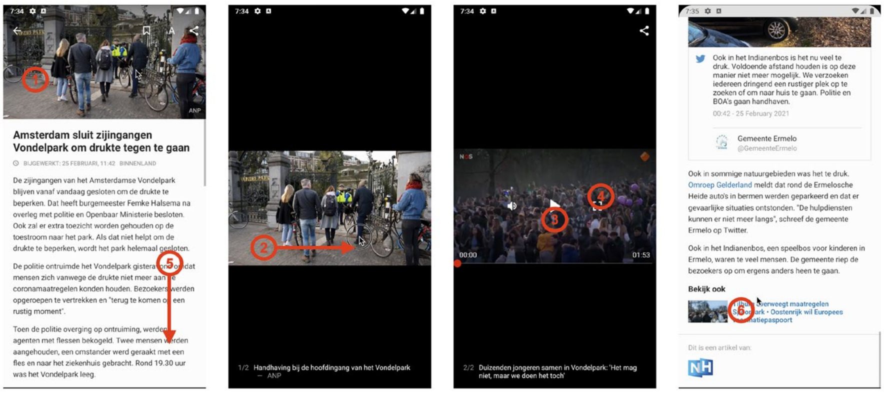
Before start, Madusa instruments the original APK file using an off-the-shelf code coverage tool, so that it can measure code coverage and resource usage of the app during the execution of the app. After the instrumentation, the user is required to write an usage scenario that exercises the desired features of a lightweight application. In particular, an input to Madusa is a specification comprising an entry point (which is optional) and a sequence of events. The followings are such an input specification to generate a simplified version of `NOS` with the same functionality as the official instant app that provides a feature of viewing one specific news article.

*Entry point:* "nl.nos.app.activity.ArticlePagerActivity" (Data for the Entry Point: "https://nos.nl/artikel/2370183-...open.html")

*Event seqence:* Touch (1) - Drag (2) - Touch (3) - Touch (3) - Touch (4) - Push Back Button - Drag (5) - Touch (6)

The input comprises two parts. First, the user needs to specify an entry point of the desired app. This information is required only if the desired app is to start from an activity other than the original main activity (the entry point is considered the main activity unless otherwise specified). In this example, we specify a specific entry point since we want the user to experience the core feature from the beginning

[2] Available at https://github.com/astean1001/madusa



**Amsterdam sluit zijingangen Vondelpark om drukte tegen te gaan**

BIJGEWERKT: 25 FEBRUARI, 11:42 BINNENLAND

De zijingangen van het Amsterdamse Vondelpark blijven vanaf vandaag gesloten om de drukte te beperken. Dat heeft burgemeester Femke Halsema na overleg met politie en Openbaar Ministerie besloten. Ook zal er extra toezicht worden gehouden op de toestroom naar het park. Als dat niet helpt om de drukte te beperken, wordt het park helemaal gesloten.

De politie ontruimde het Vondelpark gisteravond omdat mensen zich vanwege de drukte niet meer aan de coronamaatregelen konden houden. Bezoekers werden opgeroepen te vertrekken en "terug te komen op een rustig moment".

Toen de politie overging op ontruiming, werden agenten met flessen bekogeld. Twee mensen werden aangehouden, een omstander werd geraakt met een fles en naar het ziekenhuis gebracht. Rond 19.30 uur was het Vondelpark leeg.

1/2 Handhaving bij de hoofdingang van het Vondelpark — ANP

2/2 Duizenden jongeren samen in Vondelpark: 'Het mag niet, maar we doen het toch'

Ook in het Indianenbos is het nu veel te druk. Voldoende afstand houden is op deze manier niet meer mogelijk. We verzoeken iedereen dringend een rustiger plek op te zoeken of om naar huis te gaan. Politie en BOA's gaan handhaven.

00:42 - 25 February 2021

Gemeente Ermelo
@GemeenteErmelo

Ook in sommige natuurgebieden was het te druk. Omroep Gelderland meldt dat rond de Ermelosche Heide auto's in bermen werden geparkeerd en dat er gevaarlijke situaties ontstonden. "De hulpdiensten kunnen er niet meer langs", schreef de gemeente Ermelo op Twitter.

Ook in het Indianenbos, een speelbos voor kinderen in Ermelo, waren te veel mensen. De gemeente riep de bezoekers op om ergens anders heen te gaan.

**Bekijk ook**

Tilburg overweegt maatregelen Spoorpark • Oostenrijk wil Europees vaccinatiepaspoort

**Fig. 2** Visualization of the example usage scenario

(i.e., reading the news article) without going through any preparation steps (e.g., signing in to an account) as in the original app. The "Data for the Entry Point" is required only if there is additional data necessary to initialize the starting activity. In this case, we provide a link to an article that we want to show to the user.

The usage scenario as an event sequence is visualized in Fig. 2. After the starting activity launches, the event sequence exercising the core feature includes touching the thumbnail of a news video (event 1), playing and stopping the video (event 2 and 3), switching to the full screen mode (event 4), scrolling down to the bottom of the article (event 5), and clicking the link to another related article (event 6).

Ideally, such usage scenarios should extensively exercise the entire functionalities of the desired simplified apps. However, it is not always possible to devise such exhaustive event sequences. We will show how our ILP-based method can compensate for such insufficiency to the extent that the size restriction (less than 15 MB) allows.

## 2.2 Our ILP-based reducing

From the usage scenario, Madusa first measures code coverage by running the usage scenario on the original application. Madusa currently measures the coverage at the method level, which can be done by using an existing tool for measuring code coverage of Android apps (Pilgun et al. 2020), however, any code coverage tool can be used.

Next, Madusa analyzes the original app and obtains *resource usage information*, which concerns resources (e.g., bitmaps) used by each individual method. Such information can be obtained by analyzing code and XML files in the app.

Given the information of code coverage and resource usage, Madusa constructs a directed graph we call *application dependency graph* (ADG). In this graph, each node represents either a method or a resource file in the application. A node for methods (*method node*) contains information about a fully qualified method name and whether or not the method is covered (i.e., executed) while running the scenario. A node for resources (*resource node*) contains information about type and size of a resource file. Each directed edge from method node *A* to method node *B* means

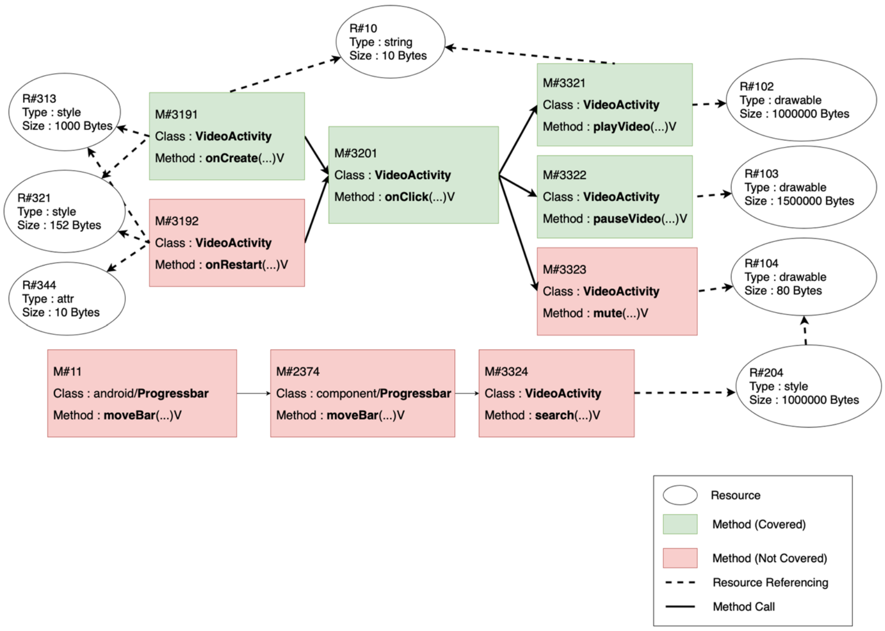


**Fig. 3** Example application dependency graph

that method *A* calls method *B*. Each directed edge from method node *A* to resource node *B* means that method *A* uses resource *B*. Each directed edge from resource *A* to resource *B* means that resource *A* references another resource *B*. Lastly, there cannot be an edge from a resource node to a method node.

Figure 3 shows a sub-part of the ADG for NOS. Each square node represents a method node, and each round node corresponds to a resource node. Dotted and solid edges represent resource uses, and method invocations, respectively. Nodes in green and red are covered and uncovered methods while running the scenario respectively.

With this information, an immedate method for generating a trimmed version of the original app is to include only methods covered by the usage scenario along with resources used by the methods. In this approach, the resulting app would include methods in green (M#3191, M#3201, M#3321, M#3322) and resources reachable from those methods (R#10, R#313, R#321, R#102, R#103).

However, this approach based solely on code coverage may lead to a brittle app which may easily crash on events never seen in the specific usage scenario. For example, the methods `onRestart` (M#3192) and `mute` (M#3323) will not be included in the resulting app, thus the app may abnormally terminate if the user presses home button and comes back to the video screen (which would invoke `onRestart`) or touches the mute button (which would invoke `mute`).

In order to enhance robustness, our key idea is to include as many methods/resources as possible while complying with the 15 MB size contraint via integer linear programming (ILP). For the given ADG, Madusa generates a trimmed version of the app by solving an ILP instance. We aim to find a *maximum* subset of nodes in the ADG sastisfying the following constraints.

- Methods covered by the usage scenario (green nodes in Fig. 3) must be included.
- If a method is included and it uses resources, all the used resources must be included.
- If a method is included and it has callers, at least one of the caller methods must be included.
- If a resource is included and it references other resources, all the referenced resources must be included.
- The sum of the sizes of included nodes must not exceed the size limit.

A solution of the ILP problem represents a set of methods and resources that should exist in the simplified app. We generate a lightweight version of the app by simply excluding methods and resources not existing in the ILP solution. In this example, methods `onRestart` and `mute` are included in the solution because the size increase caused by including them is not significant (only resources R#344 and R#104 should be additionally included at the cost of additional 90 bytes). Therefore, the app can be more robust than the one could be generated solely based on coverage.

Note that our method naturally prioritizes methods and resources closely related to covered methods. For example, method `moveBar` (M#11) has not been included since the cost outweighs the benefit. Adding the method would require to add all the transitive callees of it (M#2374, M#3324) and the used resource (R#204) which takes 1 MB. However, method `onRestart` has been included because it "shares" resources with another covered method `onCreate`.

After removing the methods and resources not to be included, we can obtain a lightweight app that provides the desired functionality while not violating the size restriction. From the 22MB-sized original app (6 MB of code, 16 MB of resources) Madusa successfully generates a 10 MB-sized (6MB of code, 4MB of resources) lightweight application within 2 h. We confirm the Madusa-generated version behaves exactly same as the official instant app of `NOS`.

## 3 Our approach

In this section, we formally describe our reducing method for mobile applications.

### 3.1 Application dependency graph

To represent mobile apps at the granularity of methods and resources, we view apps as weighted directed graphs that consist of a set of vertices with real-valued weights and a set of edges. An edge from $p$ to $q$ is denoted $p \rightarrow q$. We will write $p \rightarrow^* q$ if there exist edges leading from $p$ to $q$.

We are interested in identifying a sub-part of an original app which can be considered an *induced subgraph*. Given a subset $V'$ of $V$, an induced graph $G[V']$ is a graph whose vertex set is $V'$ and whose edge set comprises all of the edges in $E$ that have both endpoints in $V'$. In other words, for any two vertices $v_1, v_2 \in V'$, $v_1$ and $v_2$ are adjacent in $G[V']$ iff they are adjancent in $G$.

A mobile app will be represented as a graph we call application dependency graph.

**Definition 1** (*Application Dependency Graph (ADG)*) An ADG $G = \langle V, E, w \rangle$ is a directed graph where each vertex in $V \subseteq M \cup R$ is either a method in $M$ or a resource in $R$, $E \subseteq V \times V$, and $w : V \to \mathbb{R}^+$ is a function that gives a weight (i.e., size) of for a given vertex. Using $w$, we can measure the size of an ADG $G$ (denoted $|G|$) which is $\sum_{v \in V} w(v)$. For all methods and resources, the following should hold.

$$\forall m \in M, r \in R.\ m \in V' \land m \to^* r \text{ in } G \implies r \in V'.$$

The above formula says if an ADG contains a method $m$, any resources necessary for executing the method should also exist in the ADG.

We can define induced subgraphs of ADGs as follows:

**Definition 2** (*Induced ADG*) Given an ADG $G = \langle V, E, w \rangle$, an induced ADG of $G$ whose vertices are $V' \subseteq V$ is a tuple $\langle G[V'], w \rangle$ where $G[V']$ is an induced subgraph of $\langle V, E \rangle$.

## 3.2 Problem statement

Given an ADG $G = \langle V = M \cup R, E, w \rangle$ representing an original app, $V_C \subseteq V$ which is a set of vertices that must be included (i.e., methods covered while running user-provided usage scenarios), and $\theta \in \mathbb{R}^+$, our goal is to identify a *maximum induced ADG* $\langle G[V'], w \rangle$ such that

- $V_C \subseteq V'$
- $|G[V']| \leq \theta$
- $G[V']$ has as many vertices as possible.

This problem is NP-hard for the following reason.

**Theorem 1** *Given an ADG $G = \langle V, E, w \rangle$, $V_C \subseteq V$, and $\theta \in \mathbb{R}^+$, finding a maximum induced ADG of G of size $\leq \theta$ is NP-hard.*

***Proof*** We show the NP-hardness by a reduction from the problem of finding a maximum common induced subgraph of two graphs of which goal is to find an induced subgraph of both $G_1 = \langle V_1, E_1 \rangle$ and $G_2 = \langle V_2, E_2 \rangle$ and that has as many vertices as possible. Finding such a graph is NP-hard.

We can construct an ADG $G = \langle V_1 \cup V_2, E_1 \cup E_2, w \rangle$ where $w$ is defined to as follows:

**Variables:** $_v$ for each $v \in V$ with the following meaning.

$$x_v \neq 0 \iff v \in V'$$

where $V' \subseteq V$ is the set of vertices of a resulting maximum induced ADG.

**Objective:** maximize $V'$

**Subject to:**

$$\begin{array}{rll}
x_v = \overset{x}{1} & (\text{for all } v \in V_C) & (1)\\
(\sum_{m_p \text{ s.t. } m_p \to m} x_{m_p}) - x_m \geq 0 & (m, m_p \in M, \exists m_p.\ m_p \to m) & (2)\\
x_r - x_m \geq 0 & (\text{for } r \in R, m \in M, m \to r) & (3)\\
x_{r_p} - x_r \geq 0 & (\text{for } r, r_p \in R, r_p \to r) & (4)\\
(\sum_{v \in V} w(v) \cdot x_v) \leq \theta & & (5)
\end{array}$$

**Fig. 4** Given an ADG $G = \langle M \cup R, E, w \rangle$, $V_C \subseteq V$, and $\theta \in \mathbb{R}^+$, finding a maximum induced ADG $\langle G[V'], w \rangle$ by solving an ILP instance. All variables take values in $\{0, 1\}$

$$w(v) = \begin{cases} 1 & \exists v' \in V_1 \cap V_2.\ \{(v, v'), (v', v)\} \cap (E_1 \cap E_2) \neq \emptyset \\ |V_1 \cup V_2| + 1 & (\text{otherwise.}) \end{cases}$$

In other words, we assign 1 as weight only to vertices that may be included in any common induced subgraph, and the other remaining vertices are assigned a large weight. In this setting, if we find a maximum induced ADG of $G$ of size $\leq |V_1 \cup V_2|$ (where $V_C = \emptyset$), such an ADG will contain only vertices in a common induced subgraph of $G_1$ and $G_2$ (because the size constraint will be violated if any vertex exclusive to only $G_1$ or $G_2$ is chosen). And the ADG will have as many such vertices as possible. Therefore, the maximum induced ADG corresponds to the maximum common induced subgraph of $G_1$ and $G_2$, which concludes the proof. $\square$

## 3.3 Finding a maximum induced ADG via ILP

We present how to find a maximum induced ADG from a given ADG $G$ by encoding the problem into ILP. Informally, an ILP instance is a set of inequalities and equalities, where variables and constants are supposed to be integers.

**Definition 3** (*ILP*) Given are a matrix $A$ and a vector $b$. Decide whether there exists a non-negative integer vector $x$ such that $Ax \geq b$.

It is well known that the ILP problem is NP-hard (Papadimitriou 1981) as is the maximum induced ADG problem, which justifes our approach using ILP.

Figure 4 depicts our ILP encoding given an ADG $G = \langle V = M \cup R, E, w \rangle$, $V_C \subseteq V$, and $\theta \in \mathbb{R}^+$. For each vertex $v \in V$, we introduce a variable $x_v$ with values in $\{0, 1\}$. Each $x_v$ holds 1 if and only if $v$ is included in the final maximum induced ADG (i.e., $x_v \neq 0 \iff v \in V'$ where $\langle G[V'], w \rangle$ denotes the maximum induced ADG we aspire to). The constraint (1) says each vertex in $V_C$ should be included in $V'$ because they are essential to replay the usage scenarios. The constraint (2) says if a method is included, and the method has callers (i.e., predecessors in the ADG) which invoke it, at least one caller should also be in the result. This condition is to avoid adding unreachable methods into the result. The constraint (3) says a method is included, every resource used by the method should also be included. The constraint (4) says if a resource is included, every resource used by the resource should also be included. Lastly, the constraint (5) enforces the result not to exceed the size limit $\theta$.

***Example 1*** For the ADG depicted in Fig. 3, we generate the following ILP constraints where $x_{mi}$ and $x_{ri}$ denotes variables for methods and resources of ID $i$ respectively.

$$\begin{array}{ccc}
\text{From (1) in Fig. 4} & \text{From (2) in Fig. 4} & \text{From (3) and (4) in Fig. 4} \\
x_{m3191} = 1 & x_{m3191} + x_{m3192} - x_{m3201} \geq 0 & x_{m3201} - x_{m3323} \geq 0 \\
x_{m3201} = 1 & x_{m3191} + x_{m3192} - x_{m3201} \geq 0 & x_{r103} - x_{m3322} \geq 0 \\
x_{m3321} = 1 & x_{m3201} - x_{m3321} \geq 0 & x_{r103} - x_{m3322} \geq 0 \\
x_{m3322} = 1 & x_{m3201} - x_{m3323} \geq 0 & x_{r104} - x_{m3323} \geq 0 \\
\vdots & \vdots & \vdots
\end{array}$$

### 3.4 Main algorithm

**Algorithm 1** The MADUSA Algorithm

```
Input: An original app $Apk$ and a set $S$ of usage scenarios
Input: An upper bound of size of the final app $n$
Output: A simplified app $apk$ whose size is not larger than $n$
 1: $apk \leftarrow \bot$
 2: $lb \leftarrow 0$    $\theta \leftarrow n$    $ub \leftarrow$ SizeOf(UnpackAPK($Apk$))
 3: $C \leftarrow$ MeasureCodeCoverage($Apk, S$)
 4: $G \leftarrow$ GetADG($Apk, C$)
 5: repeat
 6:     $G' \leftarrow$ FindMaximumInducedADG($G, \theta$)
 7:     if $G' = \bot$ then
 8:         $lb; \leftarrow \theta$    $\theta \leftarrow (lb + ub)/2$
 9:     else
10:         $Apk' \leftarrow$ ApkOfADG($G'$)
11:         if SizeOf($Apk'$) $\leq n$ then
12:             $apk \leftarrow Apk'$
13:             $lb \leftarrow \theta;$    $\theta \leftarrow (ub + \theta)/2$
14:         else
15:             $ub \leftarrow \theta;$    $\theta \leftarrow (lb + ub)/2$
16:         end if
17:     end if
18: until time budget expires (or $lb = ub$)
19: return $apk$
```

Finding a maximum induced ADG does not suffice for obtaining the desired simplified app satisfying the size constraint. That is because an Android app is usually a single compressed container file in the APK file format into which all of code, resources, certificates, and manifest files are packaged. Because the size constraint is over the size of a final APK file rather than the sum of sizes of its parts, it is not straightforward how to determine the size limit for finding a maximum induced ADG. Our key idea to resolve this issue is to adopt binary search to find a proper size upper bound for a maximum induced ADG. Our method is based on the assumption that the compression program for packaging apps is generally monotone: the order in size between two unpackaged apps (i.e., apps not compressed into an APK file) is preserved after compression. We observe this assumption is often true in practice.

Algorithm 1 depicts our main algorithm. The algorithm takes an original app *Apk*, a set $S$ of usage scenarios, and a size upper bound $n$ of the final app as input. The goal is to generate a simplified app *apk* whose size is not larger than $n$. The result *apk* is initialized to be $\perp$ (line 1) which means a failure, and to be updated when the desired app is found. The variables $lb$ and $ub$ represent an upper bound and a lower bound of the size $\theta$ of the the desired maximum induced ADG, respectively. The variables $lb$ and $ub$ are initialized to be 0 and the sum of the sizes of all parts of the app which can be obtained by decompressing the APK file (line 2). $\theta$ is initially set to be $n$, which means we first attempt to find an ADG of size $\leq n$. Then, we measure code coverage by replaying the usage scenarios on the original app (line 3). From the code coverage information and the original app, we construct an ADG (line 4). When constructing an ADG, we statically analyze the app's code to identify relationships among different methods and resources, which will be detailed in the next section. Equipped with the ADG, the algorithm repeats the main loop (lines 5–18) until a time budget expires or the lower and upper bounds become equal. The main loop starts with obtaining a maximum induced ADG (line 6) by invoking an off-the-shelf ILP solver as already described in Sect. 3.3. No solution to the ILP instance can be found (line 7) if the size restriction is too restrictive. Then, the size limit $\theta$ is increased by taking the middle point of the upper half of the interval $[lb, ub]$ (line 8). If a solution $G'$ to the ILP instance exists (line 9), we construct a simplified version of the app by packaging only methods and resources existing in the solution into an APK file (line 10). Next, we check if the APK file satisfies the size constraint (line 11). If so, we record the app as the best result obtained so far (line 12), and increase $\theta$ similarly to line 8 (line 13), hoping to find another new app containing more methods/resources while still satisfying the size constraint. Otherwise, we decrease the upper bound by taking the middle point of the lower half of the interval $[lb, ub]$ to search for a smaller sized ADG that can satisfy the size constraint (line 15). After the main loop terminates, the algorithm returns the best result obtained so far (line 19) (if no app could be found, it returns $\perp$).

## 3.5 Implementation

In this section, we discuss noteworthy implementation details.

### 3.5.1 Call graph analysis

To construct an ADG, we identify calling relationships between methods through a call graph analysis. We perform the standard Class Hierarchy Analysis (CHA) to resolve virtual call sites. This approach may lead to an overapproximation of calling relationships.

Note that the precision of a used call graph analysis does not affect the correctness of our method. It will never generate an app violating the size constraint or unable to replay given usage scenarios. That is due to the constraint (1) in Fig. 4 and line 11 in Algorithm 1. On the other hand, the precision of a used call graph analysis may let the algorithm generate an app unnecessarily containing unreachable methods.

***Example 2*** Suppose there is a method $A$ which is determined to exist in the result (i.e., $x_A = 1$ according to an ILP solution), and there is a unique caller $B$ that invokes $A$. In addition, let us assume our CHA-based call graph analysis concludes two methods $B$ and $C$ may invoke $A$ due to its inherent imprecision. A possible solution satisfying the constraint (2) in Fig. 4 is $x_A = 1, x_B = 0, x_C = 1$, which says only methods $A$ and $C$ without $B$ are included in the result. In such a case, the $A$ method will be unreachable in the resulting app since the $C$ method is not an actual caller whereas the $B$ method, the actual caller, is missing.

If we use a more precise call graph analysis such as pointer analysis-based one for Android (Arzt et al. 2014), we can better mitigate the above problem. However, in the experiment, we note our CHA-based call graph analysis is precise enough to avoid adding too many unreachable methods and resources.

### 3.5.2 Identifying resource uses

Next, to identify dependencies between resources and methods, we conduct a simple static analysis. First we obtain all resource IDs by parsing `res/values/public.xml` that stores resource IDs. And parsing the resource xml files to identify reference relationship between resources. And for each method, we collect hard-coded resource IDs to exactly identify used resources. Additionally, through a simple intra-procedural static analysis to track possible string values for each program variable, we identify possible string arguments to the `getIdentifier` Android API method, which are used to construct a fully quantified resource name. A complication that arises here is that such string arguments may be obfuscated, which makes it difficult to identify exact resources.

For a better understanding, Listing 1 shows three cases of referencing resources. The first case shows the easiest case where the `getResourcebyId` method is invoked with a fixed resource ID which is a 8 digit number. From the ID, we can easily identify the used resource. The second case shows the necessity of tracking possible string values for each program variable. The `getIdentifier` method is invoked with string variables as arguments to identify a resource ID. To track possible string values for each program variable, we perform a simple intra-procedural analysis with the prefix abstract domain (Costantini et al. 2011) which approximates strings by their prefix. For example, an abstract domain element "*abc* $*$" represents all the strings which begin with "*abc*", including "*abc*" itself. In this case, all of the three variables have constant values, so that we can simply identify a fully quantified resource name without using the power of the prefix abstract domain. On the other hand, a complication arises when arguments to the `getIdentifier` method are obfuscated as shown in the third case. The resource name represented as variable `v3` is composed of “flag”, “-” and some string value obfuscated through a complicated expression. Through the string analysis, we infer the value of `v3` starts with “flag-”, which is represented as “flag-*” in the prefix abstract domain. In case of no prefix cannot be known as in the case of variable `v4`, the abstract value is represented as just “*” which can be any string. From this information, any resources of which name begin with “flag-” are considered to be potentially used.

In our implementation, to prevent the analysis from tracking excessively long string prefixes, we limit the maximum prefix size to be 5.

Listing 1: Threee Cases of Referencing Resources

```
// Case 1 : with a hardcoded resource ID
int v0 = 0x7f0b0024 // Resource ID
Button v1 = getResourcebyId(v0) // reference resource by id

// Case 2 : with a resource name
String v3 = "status_bar_height"
String v4 = "dimen"
String v5 = "android"
int v0 = getIdentifier(v3,v4,v5) // get the resource id of
    android:dimen/status_bar_height
float v1 = getResourcebyId(v0)

// Case 3 : with an obfuscated resource name
String v1 = "flag"
String v2 = "-"
String v3 = v1 + v2 + ... complicated expression ...
String v4 = ... complicated expression ...
Int v0 = getIdentifier(v4,v3,getPackageName())
// get the resource id of package_name:*/flag-* where * denotes a
    wildcard
Drawable v1 = getResourcebyId(v0)
```

## 4 Evaluation

We experimentally evaluated our method to answer the following research questions:

- RQ1 - Effectiveness: How effectively does Madusa reduce a given application in terms of reduction quality?
- RQ2 - Robustness: How robust is the reduced version generated by Madusa against new unseen events?

### 4.1 Setting

#### 4.1.1 Implementation

Madusa consists of 1.5K lines of Python code. We used a code coverage tool called ACVTool and GLPK, an open source ILP solver. Madusa first parses `public.xml` that has a pair of a resource ID and a resource name to obtain mapping from resource names to IDs. Then, Madusa performs a CHA-based analysis to identify relationships between methods, and invokes ACVTool to obtain code coverage information. For ILP solving, Madusa invokes GLPK to find a solution of an ILP instance. Lastly, we use APKTool Apktool (2021) to build the final debloated application from the code and resources.

All experiments were conducted on a MacBook Pro with CPUs of 2.6 GHz and 16GB memory.

#### 4.1.2 Benchmarks

Table 1 shows the characteristics of 19 applications in our benchmark suite. We first collected 235 applications that can be supported by ACVTool from CICMalDroid benign dataset (CICMalDroid 2020), F-Droid (F-Droid 2021) and Google Play Store (Google Play Store 2021). ACVTool cannot support MultiDEX applications of which code is splitted into multiple DEX (Dalvik Executable) files because ACVTool can measure coverage of a single DEX file. Furthermore, applications using native libraries through JNI are also out of the scope of ACVTool since it cannot measure coverage of native libraries. These limitations of ACVTool make it challeging to conduct experiments on large-sized applications which are often out of the scope of ACVTool. Among 235 applications, we chose 15 moderate-sized applications on which ACVTool runs successfully. In addition to this set, we also collected four applications (`nos`, `wego`, `naukri`, and `vimeo`) which officially provide Google Play Instant versions.(Lee et al. 2022)

#### 4.1.3 Baseline

We compare Madusa against a variant of Madusa (denoted Cov from now on) that generates a simplified app only based on code coverage without relying on the ILP solving. In other words, this ablation simply removes all methods and resources not used while replaying given usage scenarios. This comparison is for an ablation study to show if our ILP-based method is effective in enhancing robustness.

#### 4.1.4 Specifications

Table 2 gives the details of the usage scenarios we used for the experiments. The columns "# of Activities" and "# of UI events" show the number of activities visited and UI events exercised in the scenarios, respectively. For each application, we provide a sequence of events as an usage scenario that visits up to two screens (or windows) based on our observation about instant apps available in Google Play Store. After inspecting 38 instant apps in the Google Play Instant promotion collection, we realize they provide representative features that can be usually experienced in 1–2 screens. Based on this observation, for each application, we devise event sequences that can visit only a few screens and exercise most of the runnable features on those screens.

**Table 1** Characteristics of the benchmark apps. **Size** gives the size of an APK file

| App name | Version | Description | Size |
|---|---|---|---|
| com.**hrs**.b2c.android | 7.3.0 | Hotel search | 15.08 MB |
| com.mediadjz.**pianomixer** | 1.3 | DJ mixing app | 15.21 MB |
| nl.**nos**.app | 5.7.1 | News | 22.07 MB |
| **naukri**App.appModules.login | 12.6 | Online job search | 6.34 MB |
| com.**wego**.android | 6.0.0 | Hotels & flights booking | 11.81 MB |
| com.google.samples.apps.**topeka** | – | Quiz | 3.98 MB |
| com.**vimeo**.android.videoapp | 3.14.0 | Video viewing | 17.47 MB |
| **bbc**.mobile.news.ww | 4.0.0.80 GNL | News | 15.25 MB |
| com.royalapp.**vanlentineframes** | 1.0 | Photography | 15.18 MB |
| com.**oliveyoung** | 2.4.20 | Shopping | 13.86 MB |
| com.**huffingtonpost**.android | 26.17.0 | News | 34.37 MB |
| com.**ft**.news | 2.217.0 | News | 8.19 MB |
| com.**topten10**mall.mallapp | 1.0.0.79 | Shopping | 6.28 MB |
| de.hosenhasser.**funktrainer** | 1.3.1.4 | Quiz | 29.67 MB |
| eu.veldsoft.**complica4** | 1.3 | Game | 3.66 MB |
| link.standen.michael.**fatesheets** | 1.2 | Character sheets | 2.19 MB |
| com.games.boardgames.**aeonsend** | 1.0 | Boardgame wiki | 4.14 MB |
| com.alaskalinuxuser.just**chess** | 2.0 | Chess | 3.39 MB |
| com.daniel.mobile**pauker**2 | 2.2.0 | Flashcards | 4.80 MB |

The reader may wonder how representative the usage scenarios are of the main functionalities of the apps. Though it is difficult to quantify the complexity of usage scenarios, the ratio between the original code size and the code size of app reduced by the Cov variant can be a proxy for the complexity. In general, the more complex usage scenarios, the more code is executed. That is because the complex usage scenarios are likely to exercise a majority of features of the app. Likewise, if usage scenarios are simple, we can expect the size of an app debloated by Cov, which solely relies on the code coverage, to be small. However, even this parameter is not always accurate because there may exist a significant portion of unconditionally executed code (e.g., code for initialization and preparation steps). Therefore, we publicize recorded screens showing our usage scenarios, so that the reader can evaluate the complexity and representativeness.[3]

## 4.2 Effectiveness of reduction

We first evaluate the effectiveness of Madusa in terms of reduction size (Table 3). For each benchmark, the size limit is first set to be 50% of the original size, and incrementally increased if a solution cannot be found as already described in the

[3] Available at https://doi.org/10.5281/zenodo.7272254

main algorithm (Fig. 1). The reason for targetting 50% reduction is as follows. According to Google's internal research, when the size of the application is reduced from 10 to 5 MB, application install rate (the proportion of store visitors who install) increases by about 20%, which is a significant improvement (our benchmark apps are 12.3 MB on average). Of course larger reduction may lead to a better application install rate, but an excessively reduced application would be very likely to be brittle. Therefore, we conjecture 50% reduction strikes a good balance between apps' robustness and accessibility.

Figure 5 shows the results. It shows an average reduction rate of 39.7% and the maximum reduction rate of more than 60%. In most apps, the reduction rate for resources is higher than that for code. In `topeka`, `wego`, and `valentine-frames` since the usage scenarios exercise almost all of the apps' features, Madusa barely reduces the app sizes. However, in the other remaining apps, Madusa shows significant reduction rates. In particular, we can see a remarkable reduction for `nos`. In this app, there are many icon images taking a large portion of the entire size. Madusa removes a lot of them which are not used for replaying our usage scenario. We also note that in many cases, by removing methods unnecessary for the desired features, Madusa subsequently removes resources used by those removed methods, which leads to a significant reduction.

However, Madusa cannot significantly reduce the size of an app when the usage scenario exercises almost all the features of the app because there is not much room for reduction. Apps `naukri`, `wego`, and `vanlentineframes` fall into this category.

Also, for `huffingtonpost` and `funktrainer`, Madusa cannot significantly reduce the size. In case of `huffingtonpost`, most of the library code is used in the app, and it is difficult to remove the code. That is because all the web UI-related libraries are used immediately after the app is launched. In addition, the code of the libraries takes a large portion of the app size. That is why Madusa fails to reduce the size despite the significant reduction of the resources.

Similarly, in case of `funktrainer`, the code for loading UI components is executed immediately after the app is launched, and the code takes a significant portion of the app size.

These cases show that Madusa cannot reduce the size of an app if the code for initializing the app takes a large portion of the app size.

**Answer to Q1:** Madusa is effective in reducing the sizes of apps (with an average reduction rate of 40%).

## 4.3 Application robustness

We measure the robustness of the Madusa-generated apps by comparing against apps simplified by the Cov variant, which does not use our ILP-based method but just rely on code coverage information. We manually confirmed that applications generated by both tools work well for usage scenarios. We evaluate the robustness of the applications generated by the tools with Monkey (UI/Application Exerciser

**Table 2** Descriptions of the usage scenarios

| App | # of Activities | # of UI events | Scenario description |
|---|---|---|---|
| Hrs | 5 | 9 | Search hotels in Berlin and Newyork, go to "HRS Deals" and tab one of the hotels in deals |
| Pianomixer | 3 | 13 | Play some pianos and watch an ads |
| Nos | 3 | 11 | View a news article and videos, click links to other articles |
| Naukri | 3 | 13 | Search for front-end developer jobs in Delhi and click a search result |
| Wego | 10 | 41 | Search for flights from ICN to NRT, click a search result, search for hotels in Busan, and tab a search result |
| Topeka | 2 | 7 | Sign in and click "Food & Drink" category and come back to the main page |
| Vimeo | 3 | 14 | Play a video and touch "Share and like" button, tab another videos in the main page |
| Bbc | 3 | 9 | Go to "Video" tab, click one of the top-ranked videos to play it |
| Vanlentineframes | 4 | 8 | Turn on the camera |
| Oliveyoung | 3 | 9 | View a product page and watch product detailed info |
| Huffingtonpost | 2 | 7 | View a news article and click links to other articles |
| Ft | 4 | 12 | View a news article and swipe to watch another news articles |
| Topten10mall | 2 | 12 | View a event page and click links to a product |
| Funktrainer | 2 | 5 | Click "Betriebliche Kenntnisse" and take some quiz |
| Complica4 | 1 | 3 | Play a full game |
| Fatesheets | 2 | 13 | Touch floating button and add a character and come back to main page |
| Aeonsend | 2 | 6 | View some card info page and com back to main page |
| Justchess | 2 | 8 | Touch 2 player game and play chess 3 turns |
| Mobilepauker2 | 3 | 12 | Add a flashcard and take memory test |

**Table 3** Detailed size of reduced applications

| App | Code Size | | | Resource Size | | |
|---|---|---|---|---|---|---|
| | Original | Cov | Madusa | Original | Cov | Madusa |
| Hrs | 3.53 MB | 1.38 MB | 3.44 MB | 11.55 MB | 2.94 MB | 4.11 MB |
| Pianomixer | 2.33 MB | 701.56 KB | 2.31 MB | 12.89 MB | 3.17 MB | 3.59 MB |
| Nos | 6.16 MB | 2.00 MB | 5.90 MB | 15.91 MB | 5.38 MB | 4.04 MB |
| Naukri | 3.54 MB | 2.61 MB | 2.85 MB | 2.79 MB | 1.44 MB | 1.44 MB |
| Wego | 6.59 MB | 6.11 MB | 6.33 MB | 5.22 MB | 5.11 MB | 5.11 MB |
| Topeka | 0.67 MB | 0.63 MB | 0.64 MB | 3.31 MB | 3.17 MB | 3.17 MB |
| Vimeo | 12.19 MB | 11.18 MB | 11.69 MB | 5.27 MB | 2.38 MB | 2.39 MB |
| Bbc | 5.11 MB | 4.38 MB | 4.73 MB | 10.14 MB | 5.26 MB | 5.27 MB |
| Vanlentineframes | 0.49 MB | 0.11 MB | 0.48 MB | 14.69 MB | 13.40 MB | 13.42 MB |
| Oliveyoung | 4.80 MB | 3.24 MB | 3.33 MB | 9.07 MB | 1.70 MB | 1.76 MB |
| Huffingtonpost | 25.37 MB | 23.40 MB | 23.54 MB | 9.01 MB | 2.29 MB | 2.36 MB |
| Ft | 6.22 MB | 4.06 MB | 4.34 MB | 1.97 MB | 0.82 MB | 0.82 MB |
| Topten10 | 3.84 MB | 1.41 MB | 1.52 MB | 2.44 MB | 0.46 MB | 0.46 MB |
| Funktrainer | 19.97 MB | 18.59 MB | 18.62 MB | 9.70 MB | 0.51 MB | 0.52 MB |
| Complica4 | 1.89 MB | 0.56 MB | 0.57 MB | 1.77 MB | 0.62 MB | 0.63 MB |
| Fatesheets | 1.58 MB | 0.57 MB | 0.63 MB | 0.61 MB | 0.50 MB | 0.51 MB |
| Aeonsend | 1.49 MB | 0.45 MB | 0.48 MB | 2.66 MB | 2.43 MB | 2.44 MB |
| Chess | 1.46 MB | 0.48 MB | 0.51 MB | 1.94 MB | 0.47 MB | 0.47 MB |
| Pauker | 4.21 MB | 2.23 MB | 2.26 MB | 0.59 MB | 0.56 MB | 0.56 MB |

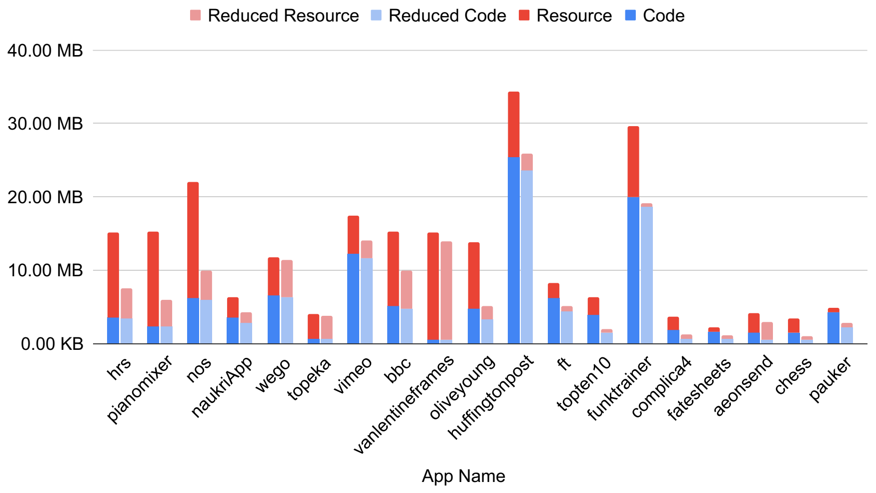


**Fig. 5** Effectiveness of Madusa in terms of reduction

Monkey 2021), which generates random streams of user events such as clicks, touches, or gestures, as well as other system-level events. For 1000 streams of user events generated by Monkey, we count how many times the applications reduced by each tool abnormally terminate (i.e., crash). We provide the same fixed seed value to Monkey to make it generate the same event sequences for both Madusa and Cov. Thus, both tools are on equal footing.

Table 4 summarizes the results. In every benchmark, the app generated by Cov crashes more often and we conclude that Madusa can effectively enhance the robustness of the resulting applications through the ILP-based method. We observe apps generated by Cov easily fail with new unseen events not existing in the provided usage scenarios.

We next investigate the results for each app in detail.

For apps `hrs`, `nos`, `pianomixer`, Madusa leads to better robustness than Cov by generating larger apps by including as many methods and resources as possible. We note the size differences bring a significant impact on the robustness of resulting apps. In particular, for `nos`, the Cov-generated version quickly crashes when the user triggers unseen events such as clicking volume control/share buttons, or swiping images. On the other hand, the Madusa-generated version contains all the methods and resources relevant to such unseen events, thereby avoiding crashes. For `hrs`, the Cov-generated app sometimes failed to fetch necessary data from the server. That is because the server worked when the trimmed version was generated whereas the server was down when we tested the app. This result shows the coverage based reducing is not robust against any nondeterministic behaviors that apps may exhibit. On the other hand, the Madusa-generated app is robust against such a situation by including error handling code (e.g., code for re-establishing the connection to the server) in the original code. In addition, the Madusa-generated app is robust against unseen events such as modifying the number of rooms, viewing available hotel lists on which the Cov-generated app crashes.

For apps `topeka`, `vimeo`, and `bbc`, Madusa shows better robustness despite marginal differences in the sizes of resulting apps by mostly including code without resources. For instance of `topeka`, the Cov-generated version crashes when the logout button is clicked. Madusa keeps the code for signing out and does not crash for the event. This enhanced robustness can be achieved with a marginal increase in size since the code size is almost negligible. Similarly, for `bbc`, the code for unseen events such as moving to other articles and playing videos is added in the Madusa-generated app on the contrary to the Cov-generated app. The `vimeo` case is also similar: various error handling code is included in the Madusa-generated without adding resources.

We also note that the Madusa-generated versions for the four applications (`nos`, `wego`, `naukri`, and `vimeo`) exhibit almost the same functionalities as their official instant apps only with marginal differences. For example, the official instant app for `naukri` provides UI components not existing in the original app and that for `wego` provides a feature for sharing news articles to others which is not supported in the Madusa-generated version.

For apps `naukri`, `chess`, the Madusa-generated version also crashes more than 10%. In the case of `naukri`, there are buttons that exercise lots of code and resources but are not exercised by the usage scenario. For example, social login buttons (which reference whole Google/Facebook authentication library and resources) and featured page buttons (which reference another page layouts) are

**Table 4** Robustness of reduced applications with their sizes.

| App | Size | | | (#Crashes / #Events) | | |
|---|---|---|---|---|---|---|
| | Original | Cov | Madusa | Original | Cov | Madusa |
| Hrs | 15.08 MB | 4.31 MB | 7.55 MB | 0% | 31% | 3.5% |
| Pianomixer | 15.21 MB | 3.88 MB | 5.90 MB | 0% | 9.3% | 1.1% |
| Nos | 22.07 MB | 7.38 MB | 9.94 MB | 0% | 60.5% | 7.1% |
| Naukri | 6.34 MB | 4.05 MB | 4.29 MB | 0% | 13.1% | 11.1% |
| Wego | 11.81 MB | 11.26 MB | 11.44 MB | 0% | 5.8% | 2.7% |
| Topeka | 3.98 MB | 3.80 MB | 3.81 MB | 0% | 23.2% | 8.5% |
| Vimeo | 17.47 MB | 14.01 MB | 14.08 MB | 0% | 15.6% | 2% |
| Bbc | 15.25 MB | 9.70 MB | 9.99 MB | 0% | 24.7% | 0.2% |
| Vanlentineframes | 15.18 MB | 13.51 MB | 13.90 MB | 0% | 8.5% | 1.6% |
| Oliveyoung | 13.86 MB | 4.94 MB | 5.10 MB | 0% | 3.8% | 2.9% |
| Huffingtonpost | 34.37 MB | 25.68 MB | 25.90 MB | 0% | 9.9% | 2.4% |
| Ft | 8.19 MB | 4.88 MB | 5.16 MB | 0% | 5.6% | 3.8% |
| Topten10 | 6.28 MB | 1.87 MB | 1.98 MB | 0% | 10.8% | 4.6% |
| Funktrainer | 29.67 MB | 19.11 MB | 19.14 MB | 0% | 0.6% | 0.3% |
| Complica4 | 3.66 MB | 1.19 MB | 1.20 MB | 0% | 0% | 0% |
| Fatesheets | 2.19 MB | 1.06 MB | 1.14 MB | 0% | 6.6% | 1.5% |
| Aeonsend | 4.14 MB | 2.89 MB | 2.92 MB | 0% | 14.7% | 7.4% |
| Chess | 3.39 MB | 0.95 MB | 0.97 MB | 0% | 11.4% | 11.4% |
| Pauker | 4.80 MB | 2.79 MB | 2.82 MB | 0% | 12.5% | 8.4% |

(#Crashes / #Events) gives the ratio between the number of crashes and the number of random event sequences generated by Monkey for each app

such buttons. These buttons take a large portion in a screen, so that many randomly generated streams of events cause the app to crash. Also, in the case of `chess` there are very few crashes during game itself. There also are some buttons such as single play button (which references whole chess ai related methods) and the single play button make up a large part of the main page, it makes our crash rate relatively high. We do not delete layout elements that are not included in usage scneario in current implementation, by Adding this feature in future might be able to lower the crash rate.

**Answer to Q2:** Apps generated by Madusa are more robust to unseen events than those directly derived from code coverage.

## 5 Threats to validity

There are several issues that cause threats to the validity or generality of our approach. We outline these next along with proposals to mitigate them.

- **Build Integrity Verification**: To improve security, android apps often adopt a *build integrity verification* to detect any changes to application source code. Because

MADUSA modifies application resources and code, it may not work for apps with this security measure. This issue can be easily mitigated by asking original developers to confirm the changes and equip the app with a new checksum.

- **Reducing Natively Compiled Libraries:** MADUSA debloats applications based on code coverage obtained at the level of dalvik bytecode, thereby being unable to obtain coverage information of natively compiled libraries that communicates with the app through JNI (Java Native Interface). For apps using popular natively compiled libraries such as Flutter or React, MADUSA may not work effectively by giving up debloating such library code. To the best of our knowledge, no known code coverage tools for Android can support native libraries. We believe this issue will be resolved by using a more advanced code coverage tool supporting native libraries in the future.
- **Exceeding 64K methods**: ACVTOOL (Pilgun et al. 2020) we use for measuring code coverage adds extra methods for instrumentation. Any coverage tools that add extra methods for instrumentation may fail when there are about 64K methods in an app. Listing 2 shows a method newly inserted by ACVTOOL for instrumentation, which is for recording whether each instruction is executed or not. It adds other several new methods to write coverage information into files. This addition may let the app exceed 65,536 methods which is the allowed limit per single DEX (Android Dalvik Executable) file. In this case, the methods should be split into multiple DEX files. Currently we cannot obtain code coverage information in such a case because ACVTOOL does not support apps with multiple DEX files. We hope this issue can be resolved by using a more advanced code coverage tool with multidex support in the future.

Listing 2: Instrumentation by ACVTool

```
// Original Android Method
class MainActivity {
    int OnPressButton() {
        some_code()
        ACVTool.isExecuted(MainActivity, OnPressButton,
            1)
        some_code()
        ACVTool.isExecuted(MainActivity, OnPressButton,
            2)
        some_code()
        ACVTool.isExecuted(MainActivity, OnPressButton,
            3)
        ...
    }
}

// Methods added by ACVTool
class ACVTool {
    void isExecuted(class_name, method_name,
       line_number) {
        CoverageStorage.save(class_name, method_name,
            line_number, true)
    }
    ...
}
```

## 6 Related work

We discuss related work on reducing mobile app sizes, program debloating, and code coverage measurement for Android.

### 6.1 App size reduction

There have been various attempts for reducing the sizes of mobile apps, but the existing approaches require developers' significant manual efforts, which has led to their sparing use. On the other hand, our method only requires usage scenarios, which can be easily provided by the users without much effort. App thinning (Anders Bertelrud 2015) in iOS is for reducing the app size by automatically detecting the user's device kind and only downloading relevant content for the device. This feature requires developers to tag their app to identify correspondences. Google's Instant App and Huawei's Quick App (Quick Apps 2021) provide a way to generate simplified versions of apps that can run without installation. Generating instant apps requires a significant amount of refactorings such as adding an instant support app bundle, modifying configurations such as a degree of network security, and adding logic for the instant experience workflow. To obtain Quick apps, developers have to create HTML5-based new apps in Javascript and CSS after learning to use a custom IDE. Proguard (Proguard 2021) is a tool for optimizing and debloating Android apps, but it aims for a different goal. Proguard preserves all the functionality of the original application by removing only unused and duplicated code. However, Madusa is more aggressive in debloating by removing even used code if it is not used in the usage scenarios, thereby removing unwanted functionality. In addition, it requires detailed processing rules specifying classes and methods that must not be discarded when shrinking apps, whereas Madusa only requires usage scenarios from the user.

### 6.2 Program debloating

Though various methods for software debloating have been proposed to automatically reduce program sizes, their goal is different from ours.

Madusa is more flexible than XDebloat (Tang et al. 2021) and ACVCut (Pilgun 2020) which also aggressively debloat android application by leaving only user-specified features. XDebloat tries to find a lightweight version of an original application that still exercise desired features specified by the user. ACVCut aims to trimming untested code based on code coverage measured from testing scenarios. On the contrary to those tools which leave minimal features necessary for satisfying the specifications, Madusa can leave extra features which are not in the specifications but helpful for making apps as robust as possible.

Xie et al. (2021), JRED (Jiang et al. 2016) and RedDroid (Jiang et al. 2018) soundly trim unused methods and classes from Java and Android applications. Quach et al. (2018) also use a sound static analysis to identify only necessary subparts of libraries based on function-level dependencies. The goal of these sound debloating methods is to reduce program sizes without losing any existing features of original programs, whereas Madusa aims at more aggressive size reduction by leaving only representative features of original programs. Heo et al. (2018) and Qian et al. (2019) are similar to our approach in that they also aggressively debloat programs by leaving only features necessary for satisfying user-provided specifications. Chisel tries to find a minimal version of an original program that still exercise desired functionalities specified by the user. Razor aims to obtain a minimal version of an original binary executable based on a given set of test cases and control-flow-based heuristics. However, the difference between Madusa and those tools is that Madusa generates lightweight versions *smaller than a size limit* whereas they try to find *minimal* programs. This difference leads to different methods for guaranteeing robustness of simplified versions. Madusa aims at finding apps of which sizes are as close to a given size limit as possible by adding existing components of the original app as many as possible. On the other hand, Chisel adopts a feedback loop using static and dynamic analysis tools to improve robustness. For a generated simplified version, it identifies any potential bugs and add constraints to avoid the bugs in the specification, hoping to generate more robust programs in the next iterations.

### 6.3 Android code coverage

Though Madusa currently uses ACVTool which works at the level of methods and `smali`, the potential user of Madusa can freely choose other coverage tool at other different levels of languages or granularities. Jacoco (2021) measures code coverage for Java and Kotlin programs, so that it can be potentially used if app debloating is to be done in the source code level. However, in such a case, Madusa may not be able to trim third-party libraries without sources, which is why we choose to work at the `smali` level. There are also other code coverage tools for `smali` such as ELLA (ELLA 2021), InsDal (Liu et al. 2017), CovDroid (Yeh and Huang 2015), and COSMO (Romdhana et al. 2021) Madusa can potentially use.

## 7 Conclusion

We have presented Madusa that adopts an ILP-based algorithm for reducing Android apps into demo app. Our approach is shown to effectively remove redundant code and resources with respect to given usage scenarios. Our ILP-based algorithm is effective in improving the robustness of reduced versions by including as many code and resources as possible while complying with a constraint over the size of the result. Our method is general in that it is potentially applicable to another OS for mobile devices such as iOS only if a proper code coverage tool is available.

**Acknowledgements** The first two authors major in Bio Artificial Intelligence at Hanyang University. This research was supported by the National Research Foundation of Korea (NRF) grant (No.

2020R1C1C1014518, 2021R1A5A1021944) and Institute for Information & communications Technology Promotion (IITP) grant (No. 2021-0-00758) funded by the Korea government (MSIT).

## Declarations

**Conflict of interest** The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## References

Anders Bertelrud, P.H.: App Thinning in Xcode. https://developer.apple.com/videos/play/wwdc2015/404/. [Online; accessed 01-October-2021] (2015)

Android Instant Apps - Android SDK statistics. https://www.appbrain.com/stats/libraries/details/instant-apps/android-instant-apps. [Online; accessed 01-October-2021] (2021)

Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps. https://ibotpeaches.github.io/Apktool/. [Online; accessed 01-October-2021] (2021)

Apps to Try Now - Android Apps on Google Play. https://play.google.com/store/apps/collection/promotion_3002d0f_instantapps_featuredapps. [Online; accessed 01-October-2021] (2021)

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Not. **49**(6), 259–269 (2014)

CICMalDroid 2020. https://www.unb.ca/cic/datasets/maldroid-2020.html. [Online; accessed 01-October-2021] (2020)

Costantini, G., Ferrara, P., Cortesi, A.: Static analysis of string values. In: Qin, S., Qiu, Z. (eds.) Formal methods and software engineering, pp. 505–521. Springer, Berlin, Heidelberg (2011)

ELLA: a tool for binary instrumentation of android apps. https://github.com/saswatanand/ella. [Online; accessed 01-October-2021]

F-Droid - Free and open source android app repository. https://f-droid.org/. [Online; accessed 01-October-2021]

Github - smali. https://github.com/JesusFreke/smali. [Online; accessed 01-October-2021] (2021)

Google Play Instant Developer Success Stories. https://developer.android.com/topic/google-play-instant#developer-success-stories. [Online; accessed 01-October-2021] (2021)

Google Play Store. https://play.google.com/store/apps. [Online; accessed 01-October-2021]

Henderson, M., Glick, K., Ng, K., Nikolic, M.: The future of apps on Android and Google Play: Modular, instant, and dynamic (Google I/O '18). https://www.youtube.com/watch?v=0raqVydJmNE. [Online; accessed 01-October-2021] (2018)

Heo, K., Lee, W., Pashakhanloo, P., Naik, M.: Effective program debloating via reinforcement learning. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. CCS '18, pp. 380–394. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3243734.3243838

Jacoco - Java code coverage for eclipse. https://www.jacoco.org/. [Online; accessed 01-October-2021]

Jiang, Y., Bao, Q., Wang, S., Liu, X., Wu, D.: Reddroid: Android application redundancy customization based on static analysis. In: 2018 IEEE 29th international symposium on software reliability engineering (ISSRE), pp. 189–199 (2018). https://doi.org/10.1109/ISSRE.2018.00029

Jiang, Y., Wu, D., Liu, P.: Jred: Program customization and bloatware mitigation based on static analysis, In 2016 IEEE 40th annual computer software and applications conference (COMPSAC), pp. 12–21 (2016). https://doi.org/10.1109/COMPSAC.2016.146

Lee, J., Cho, H., Lee, W.: Artifact of MADUSA: Mobile Application Demo Generation based on Usage Scenarios. Zenodo (2022). https://doi.org/10.5281/zenodo.7272254

Liu, J., Wu, T., Deng, X., Yan, J., Zhang, J.: Insdal: A safe and extensible instrumentation tool on dalvik byte-code for android applications. In: 2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER), pp. 502–506 (2017). https://doi.org/10.1109/SANER.2017.7884662

Papadimitriou, C.H.: On the complexity of integer programming. J. ACM **28**(4), 765–768 (1981). https://doi.org/10.1145/322276.322287

Pilgun, A.: Don't trust me, test me: 100% code coverage for a 3rd-party android app. In: 2020 27th Asia-Pacific Software Engineering Conference (APSEC), pp. 375–384 (2020). IEEE

Pilgun, A., Gadyatskaya, O., Zhauniarovich, Y., Dashevskyi, S., Kushniarou, A., Mauw, S.: Fine-grained code coverage measurement in automated black-box android testing. ACM Trans. Softw. Eng. Methodol. (TOSEM) **29**(4), 1–35 (2020)

Proguard – The Java optimizer for Android apps. https://www.guardsquare.com/proguard. [Online; accessed 01-October-2021] (2021)

Qian, C., Hu, H., Alharthi, M., Chung, P.H., Kim, T., Lee, W.: RAZOR: A framework for post-deployment software debloating. In: 28th USENIX security symposium (USENIX Security 19), pp. 1733–1750. USENIX Association, Santa Clara, CA (2019). https://www.usenix.org/conference/usenixsecurity19/presentation/qian

Quach, A., Prakash, A., Yan, L.: Debloating software through piece-wise compilation and loading. In: 27th USENIX security symposium (USENIX Security 18), pp. 869–886. USENIX Association, Baltimore, MD (2018). https://www.usenix.org/conference/usenixsecurity18/presentation/quach

Quick Apps - HUAWEI Developers. https://developer.huawei.com/consumer/en/huawei-quickApp/. [Online; accessed 01-October-2021]

Romdhana, A., Ceccato, M., Georgiu, G., Merlo, A., Tonella, P.: Cosmo: Code coverage made easier for android. In 2021 14th IEEE conference on software testing, verification and validation (ICST) (2021). https://doi.org/10.1109/ICST49551.2021.00053

Tang, Y., Zhou, H., Luo, X., Chen, T., Wang, H., Xu, Z., Cai, Y.: Xdebloat: Towards automated feature-oriented app debloating. IEEE Trans. Softw. Eng. (2021)

The iPhone's Top Apps Are Nearly 4x Larger Than Five Years Ago. https://sensortower.com/blog/ios-app-size-growth-2021. [Online; accessed 01-October-2021] (2021)

UI/Application exerciser monkey. http://developer.android.com/tools/help/monkey.html. [Online; accessed 01-October-2021] (2021)

Xie, Q., Gong, Q., He, X., Chen, Y., Wang, X., Zheng, H., Zhao, B.: Trimming mobile applications for bandwidth-challenged networks in developing regions. IEEE Trans. Mobile Comput. (2021)

Yeh, C.-C., Huang, S.-K.: Covdroid: A black-box testing coverage system for android. In: 2015 IEEE 39th annual computer software and applications conference, vol. 3, pp. 447–452 (2015). https://doi.org/10.1109/COMPSAC.2015.125