



DMMPP: Constructing Dummy Main Methods for Android Apps with Path-Sensitive Predicates

Baoquan Cui

State Key Lab. of Computer Science
Institute of Software, Chinese
Academy of Sciences (CAS)
University of CAS
Beijing, China
cuibq@ios.ac.cn

Jiwei Yan*

Tech. Center of Softw. Eng.
Institute of Software, CAS
Beijing, China
yanjiwei@otcaix.iscas.ac.cn

Jian Zhang*

Key Lab. of System Softw., CAS
State Key Lab. of Computer Science
Institute of Software, CAS
University of CAS
Beijing, China
zj@ios.ac.cn

Abstract

Android is based on an event-driven model, which hides the main method, and is driven by the lifecycle methods and listeners from user interaction. FlowDroid, constructs a dummy main method statically emulating the lifecycle methods. The dummy main method has been widely used by FlowDroid and also other Android analyzers as their entry points. However, the existing dummy main method is not designed for path-sensitive analysis, whose paths may be unsatisfiable. Thus, when using original dummy main methods, path-sensitive analysis, e.g., symbolic execution, may suffer from infeasible paths. In this paper, we present DMMPP, the first dummy main method generator for Android applications with path-sensitive predicates, and the corresponding path condition is satisfiable. DMMPP constructs dummy main methods for the four types of components in an application with a more realistic simulation for the lifecycle methods. The experiment demonstrates the benefits of our tool for path-sensitive analyzers, improving 28.5 times more explored paths with a low time overhead.

CCS Concepts

• Theory of computation → Program analysis.

Keywords

Android, Dummy Main, Entry Point, Predicate, Path-sensitive

ACM Reference Format:

Baoquan Cui, Jiwei Yan, and Jian Zhang. 2024. DMMPP: Constructing Dummy Main Methods for Android Apps with Path-Sensitive Predicates. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*, September 16–20, 2024, Vienna, Austria. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.1145/3650212.3685302>

1 Introduction

Over the past decade, Android smart devices have become an indispensable part of our lives. The Android system and its applications are widely used in smartphones, tablets, TVs, robots, cars

*Corresponding Authors.



This work is licensed under a Creative Commons Attribution 4.0 International License.

ISSTA '24, September 16–20, 2024, Vienna, Austria

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0612-7/24/09

<https://doi.org/10.1145/3650212.3685302>

and other devices due to its open source nature, rich application ecosystem, hardware compatibility and user customizability.

In recent years, researchers have conducted extensive and in-depth research on Android applications in various aspects. Unlike Java programs, an Android application does not have a main method, and the application is driven by various events, such as lifecycle methods and callbacks of user operations, from the Android operating system and framework. FlowDroid, a precise context, flow, field, object-sensitive and lifecycle-aware taint analysis framework for Android apps [3], constructs a dummy main method as the entry point for analysis, providing infrastructure for many works [17]. The dummy main method simulates multiple entry points statically, such as Activity, Service, BroadcastReceiver, ContentProvider and Fragment, asynchronously executing components and callbacks from user operations.

Since FlowDroid uses the IFDS algorithm, a path-insensitive approach, the dummy main method generated by FlowDroid does not consider whether the path condition is satisfied. However, for path-sensitive analysis [12, 16], it can affect the analysis accuracy, resulting in false positives and false negatives. For example, predicates generated by FlowDroid for a dummy main method are $p_1(i == 0)$, $p_2(i == 1)$, and $p_3(i == 2)$, whose satisfiability depends on the same variable i , which is path-insensitive. When a path condition in the method is $p_1 \wedge p_2 \wedge p_3$, it is obviously unsatisfiable, causing the lifecycle method unreachable.

In this paper, we propose DMMPP to construct the dummy main method for Android applications with path-sensitive predicates, which can form a satisfiable path condition. It takes an APK and the lifecycle specifications of components as inputs, built on the top of FlowDroid. It models the lifecycle of components with a unified lifecycle graph. For each component, DMMPP constructs the dummy main method with path-sensitive predicates after the lifecycle method complementing, according to its lifecycle graph, the instrumentation syntax and the generation algorithm. DMMPP can be used by analyzers with both path-sensitive approaches and path-insensitive ones, intrusively via an API invocation or non-intrusively via the APK output after a command line call.

2 Background

We will introduce the dummy main method and use an example to show the path-insensitive predicates generated by FlowDroid and how DMMPP refines the predicates for path-sensitive analyzers.

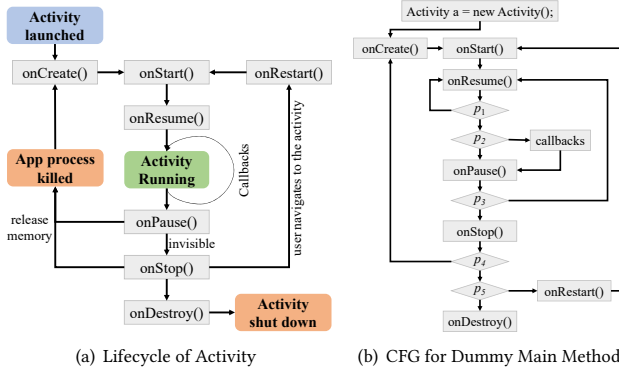


Figure 1: Activity's Lifecycle and CFG of Its Dummy Main Method Modelled by FlowDroid

2.1 Dummy Main Method

Android is based on an event-driven model, which hides the main method, and is driven by the lifecycle methods and listeners from user interaction. FlowDroid statically models the lifecycle of the component in the Android framework with the dummy main method as the entry point for its taint analysis and also for other analyzers. Figure 1 shows the lifecycle of an Activity and the control flow graph (CFG) of its dummy main method modelled by FlowDroid. During the construction of the dummy main method, FlowDroid proposes opaque predicates to represent the branches in the method, denoted as p_i , where $i \in \{1, 2, 3, 4, 5\}$.

2.2 Motivating Example

The opaque predicate, p_i , represents whether the integer variable `intCounterVar` equals to an integer `conditionCounter`. Each time an opaque predicate is inserted by FlowDroid, the `conditionCounter` is incremented by 1, i.e., `conditionCounter++` [1]. Figure 2 shows the simplified code snippet with predicates in the dummy main method generated by FlowDroid (lines 2-8), which is path-insensitive. All predicates share a single variable i , then a path condition consisting of them will be unsatisfiable. When collecting a path with the constraint, $p_1(i == 0) \wedge p_2(i == 1) \wedge p_3(i == 2)$, a path-sensitive analyzer will discard this path since the constraint is unsatisfiable. This modelling is invalid for the path-sensitive analysis since lifecycle methods are reachable definitely.

```

1 // Path-insensitive Predicates modelled by FlowDroid
2 void dummyMain(){
3     int i = 0;
4     if(i == 0){ // p1 : i == 0
5         if(i == 1){ // p2 : i == 1
6             if(i == 2){ // p3 : i == 2
7                 ...
8             }
9         }
10    }
11    // Path-sensitive Predicates
12    void dummyMain(boolean[] bArr)
13    if(bArr[0]){ // p1 : bArr[0]
14        if(bArr[1]){ // p2 : bArr[1]
15            if(bArr[2]){ // p3 : bArr[2]
16                ...
17            }
18        }
19    }
20 }

```

Figure 2: Predicates in Dummy Main Method

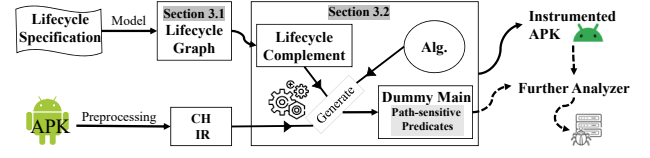


Figure 3: Overview of DMMP

The goal of DMMP in this paper is to construct the crystal predicates for the dummy main method to refine the path-insensitive ones introduced by FlowDroid for the path-sensitive analyzers. DMMP uses the boolean variables from parameters representing the predicates. To reduce the number of parameters, DMMP wraps the parameters with an array container. The dummy main method with path-sensitive predicates generated by DMMP is shown in Figure 2 (lines 10-15). Whether the path condition, $p_1 \wedge p_2 \wedge p_3$, is satisfiable will depend on the elements in the input boolean array `bArr`. The path-sensitive predicates will navigate the path exploration in the lifecycle of a component to benefit static analysis.

3 DMMP

Figure 3 shows the architecture of DMMP. It is built on the top of FlowDroid and its intermediate representation (IR) Jimple, and requires an APK and the lifecycle specifications of components as inputs. It models the lifecycle of components with a unified lifecycle graph. With the IR and the class hierarchy (CH), DMMP can recognize the components in the APK. For each component, DMMP complements its non-explicitly inherited lifecycle method to explicitly indicate component state and constructs the dummy main method with path-sensitive predicates after the complement, according to its lifecycle graph, the syntax and the generation algorithm. A path-sensitive analyzer can use the dummy main method generated by DMMP directly or use the APK after the persistence indirectly.

3.1 Model

For the convenience of expression, we formalize a component lifecycle graph with corresponding methods, and then show how to use it as input to generate the dummy main method with the instrumentation syntax.

DEFINITION 3.1.1. Component Lifecycle Graph (CLG). A component lifecycle graph is a directed graph, denoted as

$$CLG = \langle N, E, \top, \perp \rangle$$

where

- N is a set of nodes and a node represents a lifecycle method or a `nop` instruction;
- $E \subseteq N \times N$, E is a set of edges;
- $\top \in N$, \top is the start node;
- $\perp \in N$, \perp is the end node.

We assume that CLG contains a unique \top node and a unique \perp node, and for any node N_i in CLG, there exist directed paths from \top node to N_i and from N_i to \perp . We call a node a join node if it has more than one predecessor and a branch node if it has more than one successor. We also assume that a branch node has only two successors. If a lifecycle graph has a node with more than two

Name:	$\frac{A \in (a...zA...Z\$<>)^*}{name(A)}$	Type:	$\frac{name(T)}{type(T)}$
Variable:	$\frac{type(T) \quad name(V)}{var(T, V)}$	ArrayType:	$\frac{type(T)}{arrayType(T)}$
Expression:	$\frac{instruction(E)}{instruction(exp(E))}$	Return:	$\frac{-}{instruction(returnVoid())}$
Program:	$\frac{instruction(I) \quad program(P)}{program(I, P)}$	New:	$\frac{type(T)}{instruction(new(T))}$
IfStmt:	$\frac{exp(E) \quad type(B) \text{ is boolean}}{instruction(ifStmt(B, E))}$	Goto:	$\frac{exp(E)}{instruction(Goto(E))}$
Load:	$\frac{var(Index, ARR, Ret) \quad ArrayType(ARR) \quad type(Index) \text{ is integer}}{instruction(load(ARR, Index, Ret))}$		
Class:	$\frac{name(Name) \quad type(Super) \quad var(Field) \quad fun(Sign)}{class(Name, Super, Field, Sign)}$		
Function:	$\frac{name(Sign) \quad var(Arg) \quad program(Body)}{fun(Sign, Arg, Body)}$		
Invocation:	$\frac{var(Base) \quad name(Sign) \quad var(Arg) \quad var(Ret)}{instruction(call(Base, Sign, Arg, Ret))}$		
InsertExpr:	$\frac{exp(E_1) \quad fun(Sign) \quad exp(E_2)}{insertExpr(Sign, E_1, [after], site(E_2))}$		

Figure 4: Syntax for Generation.

successors, a nop instruction node is introduced and inserted to split the node. For example, the node *onStop()* in Figure 1(a) has three successors, when constructing the CLG, it should be split with a nop instruction node to ensure that each branch node has two successors like the node *onStop()* in Figure 1(b).

3.2 Generation

With the CLG, the lifecycle of each component, such as Activity, Service, BroadcastReceiver, Content Provider and even Fragment, can be expressed uniformly. Since the lifecycle methods of a component are closely related to its state [11], and some bug detection depends on its state [6, 9, 15], if a lifecycle method is not explicitly inherited, DMPPP will complement the method with the parent method invocation, such as `void onDestroy(){super.onDestroy();}`. DMPPP obtains the components and callbacks based on FlowDroid by parsing the XML resource files from the APK.

Syntax. DMPPP uses the syntax shown in Figure 4 to process the dummy main method. For each rule in it, the part below the horizontal line is the operation command and the above one is the restriction. The basic rules, such as **Name**, **Type**, **ArrayType**, **Return**, **Program**, **Expression**, **New**, **Class**, **Function**, **Invocation** and **InsertExpr**, are easy to understand. DMPPP uses the instructions **IfStmt** and **Goto** to express the branch and the loop in the CLG. In particular, for the path-sensitive predicates, DMPPP needs the operation **Load** to obtain the *i*-th element of the predicate array.

Construction. With the syntax, DMPPP constructs the dummy main method for each component according to the generation algorithm shown in Algorithm 1 based on the IR Jimple. It takes a component and its CLG as inputs. Firstly, DMPPP instantiates the component, which is the caller of the lifecycle methods in the CLG (line 4). Then, it traverses the CLG in depth-first order with a stack, generating an invocation statement for each node with the lifecycle method (lines 6-20). The actual arguments required in the invocation statement are also taken from the formal parameters of the dummy main method, simplified as *getArgsFromParameters(...)* (line 14). Thirdly, DMPPP generates the branch statement and the goto statement according to the CLG and inserts them at the corresponding position (lines 24-40). For the branch statement, DMPPP loads the predicate from the parameter array with an index, which can be assigned the value true or false by a path-sensitive analyzer to

Algorithm 1: Dummy Main Method Generation

```

Input: component, clg
1 fun ← NULL, args ← ∅, body ← ∅, node ← clg.⊤, stack ← ∅;
2 caller ← NULL, map ← ∅; // map(Node, Expression)
3 stack.push(node);
4 caller ← instruction(new(type(component)));
5 // generate method invocations
6 while stack is not empty do
7   node ← stack.pop();
8   stmt ← NULL;
9   if node is ⊥ then
10    stmt ← instruction(returnVoid);
11  else if node is not ⊤ then
12    if node.m is nop instruction then
13      stmt = instruction("nop");
14    list ← getArgsFromParameters(node.m, args);
15    stmt ← instruction(call(caller, node.m, list, NULL));
16    map.put(node, call);
17  body.add(stmt);
18  map.put(node, stmt);
19  successors ← clg.successorOf(node);
20  stack.push(successors);
21 // insert if- and goto- statements
22 predicateIndex ← 0;
23 predicates ← getPredicatesFromParameters(args);
24 for node ∈ clg.N do
25   current ← map.get(node);
26   next ← body.getNext(current);
27   if node is branch node then
28     // crystal predicate
29     load(predicates, predicateIndex, predicate);
30     predicateIndex ++;
31     for succ ∈ clg.successorOf(node) do
32       if next is not succ then
33         next ← succ;
34     ifStmt ← instruction(ifStmt(predicate, next));
35     insertExpr(body, ifStmt, "after", current);
36   succ ← clg.successorOf(node);
37   next ← map.get(succ);
38   if current is not ⊥ ∧ next is not NULL then
39     gotoStmt ← instruction(Goto(next));
40     insertExpr(body, gotoStmt, "after", current);
41 fun ← fun("dummyMainMethod", args, body);
42 return fun;

```

control the exploration along the different branches or paths. Finally, it returns the dummy main method, which will be added into the component class as an entry method for the analyzer directly (lines 41-42). DMPPP also instruments the callbacks declared in the component into the dummy main method, such as click listeners, key event listeners and so on. In addition, DMPPP can also write the generated IR back to the APK with the support of FlowDroid as its output, benefiting the path-sensitive static analyzer indirectly.

Table 1: Benefits for Analyzer

App	#C	Explored Paths			Construction Time (ms)			
		FL	D	Δ	FL	D	Δ	
F-Droid	app.fedila	6	6	12	+6	235	2960	2725
	ch.bailu.a	17	17	37	+20	414	2,038	1,624
	com.asdoi	6	6	12	+6	72	2,262	2,190
	com.gimran	13	13	30	+17	199	421	222
	com.mobile	14	14	153	+139	1,111	516	-595
	com.tuyafe	5	5	10	+5	30	1,724	1,694
	com.ubersp	1	1	13	+12	40	5	-35
	jp.takke.c	7	7	14	+7	42	1,932	1,890
	net.gitsai	8	8	2,104	+2,096	2,775	534	-2,241
	xyz.myachi	8	8	1,588	+1,580	6,991	2,377	-4,614
Google Play	appnewness	7	7	35	+28	191	554	363
	com.alb.pl	1	1	5	+4	12	4	-8
	com.e.ulil	5	5	10	+5	32	460	428
	com.pinayr	11	11	293	+282	2,094	187	-1,907
	f.fajrak.b	9	9	46	+37	278	517	239
	it.discors	9	9	49	+40	306	998	692
	kick.wpapp	8	8	40	+32	240	486	246
	kr.ieodo.a	5	5	10	+5	27	530	503
	net.easyjo	7	7	14	+7	42	1,427	1,385
	usd.aleavt	3	3	99	+96	459	59	-400
Total	150	150	4,574	+4,424	15,590	19,991	4,401	

3.3 Usage

DMMP is open source, and its latest version and executable JAR file are publicly available¹. There is also a video demonstrating DMMP with the YouTube link on Github. DMMP can be used as an API library intrusively, which will be a precursor of an Android static analyzer based on Soot/FlowDroid. It can also be used as an independent, command line tool non-intrusively to generate the dummy main method with the path-sensitive predicates and output the APK with the dummy main method, which can be used for other Android static analyzers or may be converted to a JAR file as an input of a Java static analyzer. DMMP has been integrated into Androlic [10], which is an extensible flow, context, object, field, and path-sensitive static analysis framework for Android and DMMP can also be used for other analyzers.

4 Evaluation

To evaluate the effectiveness of DMMP, we randomly collect 20 real-world apps, 10 of which are from F-Droid [4] and the other 10 from Google Play [7], which is also publicly available at Github. We choose Androlic to observe the benefits brought by DMMP, since its approach is path-sensitive. We take the dummy main method for each component generated by FlowDroid (denoted as 'FL' in Table 1) and the one generated by DMMP (denoted as 'D') as its inputs respectively. The configuration of Androlic contains a maxLoopUnrollNumber with a value of 1 and a timeout threshold of 5 minutes. The experiment is conducted under the environment of JDK-1.8, where the operating system is Windows 11 with 4 cores (Intel (R) Core(TM) i7-10510U) and 32G RAM.

Table 1 shows the number of components (denoted as "#C"), the number of feasible paths and the construction time for dummy main methods, excluding the parsing time of the application. The

symbol ' Δ ' represents the increase in the number of feasible paths and the time overhead brought by DMMP.

It can be seen that DMMP benefits the path-sensitive analyzer with about 28.5 (4,424/150 - 1) times more feasible paths than FlowDroid from the dummy main method with a time overhead of only 4.4 seconds totally. FlowDroid contributes so few feasible paths in its dummy main method because "We may skip the complete component" [2], i.e., `if(i==0){ return; }else{ lifecycle method invocations }`. For each component, the construction time of FlowDroid for the dummy main method is about 104 milliseconds, and DMMP contributes an additional 28.5 (4,424/150 - 1) times feasible paths with a time consumption 29.3 milliseconds on average. DMMP takes more time because it complements lifecycle methods and constructs a more complete dummy main method.

In summary, the evaluation demonstrates that DMMP can benefit the path-sensitive analyzer via the dummy main method generated by it with the path-sensitive predicates, providing many feasible paths with a very low time overhead as expected.

5 Related Work

The most related work to the dummy main method generation is FlowDroid. The dummy main method provided by it works for the path-insensitive analysis, such as the flow-sensitive analysis, but hinders the path-sensitive analysis, such as symbolic execution, since the predicates in a path condition can not be satisfied which violates the reachability of lifecycle methods. Compared to FlowDroid, DMMP complements the non-explicitly inherited lifecycle methods and provides the path-sensitive predicate, which can satisfy the reachability of lifecycle methods as every lifecycle method is reachable declared by Android platform. It works for both path-sensitive analysis approaches and path-insensitive ones. A review has found that analysis approaches for Android applications can be improved with more precise techniques to make them more applicable [14], which path-sensitive analysis can do. Therefore, it can help the path-sensitive work, such as taint analysis of arrays [8], malicious application detection [18], security vulnerability detection [13] and estimation of API calls [5].

6 Conclusion

We have introduced DMMP, which constructs the dummy main method with path-sensitive predicates for Android application analysis. It provides multiple ways for analyzers to use it. The experiment demonstrates the benefits of DMMP for path-sensitive analyzers. Directions for future work include using DMMP for analyzers to find real bugs in APKs and merging the same parameters from different methods/callbacks to reduce the parameter redundancy of the dummy main method.

Acknowledgement

Thanks to Dr. Linjie Pan for the initial discussion on this work, to Ms. Yajun Zhu for comments on earlier drafts of this paper, and to the anonymous reviewers for their helpful comments and suggestions. This work is supported by the National Natural Science Foundation of China (NSFC) under grant number 62132020 and 62102405, and Major Project of ISCAS (ISCAS-ZD-202302).

¹<https://github.com/cuixiaoyi/DMMP>

References

- [1] Steven Arzt. 2023. *Implementation of the Opaque Predicate in FlowDroid*. Retrieved Jul 27, 2024 from <https://github.com/secure-software-engineering/FlowDroid/blob/develop/soot-infoflow/src/soot/jimple/infoflow/entryPointCreators/BaseEntryPointCreator.java#L960>
- [2] Steven Arzt. 2023. *We may skip the complete component*. Retrieved Jul 27, 2024 from <https://github.com/secure-software-engineering/FlowDroid/blob/develop/soot-infoflow-android/src/soot/jimple/infoflow/android/entryPointCreators/components/AbstractComponentEntryPointCreator.java#L186>
- [3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Ochteau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. ACM, 259–269. <https://doi.org/10.1145/2594291.2594299>
- [4] Ciaran Gultnieks. 2010. *F-Droid: Free and Open Source Software*. Retrieved Jul 27, 2024 from <https://f-droid.org/>
- [5] Wenhao Fan, Daishuai Zhang, Ye Chen, Fan Wu, and Yuan'an Liu. 2020. EstiDroid: Estimate API Calls of Android Applications Using Static Analysis Technology. *IEEE Access* 8 (2020), 105384–105398. <https://doi.org/10.1109/ACCESS.2020.3000523>
- [6] Umar Farooq, Zhijia Zhao, Manu Sridharan, and Iulian Neamtiu. 2020. LiveDroid: identifying and preserving mobile app state in volatile runtime environments. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 160:1–160:30. <https://doi.org/10.1145/3428228>
- [7] GooglePlay. 2024. *Google Play Store*. Retrieved Jul 27, 2024 from <https://play.google.com/>
- [8] Assad Maalouf and Lunjin Lu. 2021. Taint analysis of arrays in Android applications. In *SAC '21: The 36th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, Republic of Korea, March 22–26, 2021*. ACM, 893–899. <https://doi.org/10.1145/3412841.3441964>
- [9] Linjie Pan, Baoquan Cui, Hao Liu, Jiwei Yan, Siqi Wang, Jun Yan, and Jian Zhang. 2020. Static asynchronous component misuse detection for Android applications. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8–13, 2020*. ACM, 952–963. <https://doi.org/10.1145/3368089.3409699>
- [10] Linjie Pan, Baoquan Cui, Jiwei Yan, Xutong Ma, Jun Yan, and Jian Zhang. 2019. Androlic: an extensible flow, context, object, field, and path-sensitive static analysis framework for Android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15–19, 2019*. ACM, 394–397. <https://doi.org/10.1145/3293882.3339001>
- [11] Android Platform. 2024. *Activity state changes*. Retrieved Jul 27, 2024 from <https://developer.android.com/guide/components/activities/state-changes>
- [12] Hong Song, Dandan Lin, Shuang Zhu, Weiping Wang, and Shigeng Zhang. [n. d.]. ADS-SA: System for Automatically Detecting Sensitive Path of Android Applications Based on Static Analysis.
- [13] Junwei Tang, Ruixuan Li, Kaipeng Wang, Xiwu Gu, and Zhiyong Xu. 2020. A novel hybrid method to analyze security vulnerabilities in android applications. *Tsinghua Science and Technology* 25, 5 (2020), 589–603.
- [14] Tianyong Wu, Xi Deng, Jun Yan, and Jian Zhang. 2019. Analyses for specific defects in android applications: a survey. *Frontiers Comput. Sci.* 13, 6 (2019), 1210–1227. <https://doi.org/10.1007/S11704-018-7008-1>
- [15] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An Empirical Study of Functional Bugs in Android Apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17–21, 2023*. ACM, 1319–1331. <https://doi.org/10.1145/3597926.3598138>
- [16] Sen Yang, Sen Chen, Lingling Fan, Sihan Xu, Zhanwei Hui, and Song Huang. 2023. Compatibility Issue Detection for Android Apps Based on Path-Sensitive Semantic Analysis. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14–20, 2023*. IEEE, 257–269. <https://doi.org/10.1109/ICSE48619.2023.00033>
- [17] Junbin Zhang, Yingying Wang, Lina Qiu, and Julia Rubin. 2022. Analyzing Android Taint Analysis Tools: FlowDroid, Amandroid, and DroidSafe. *IEEE Trans. Software Eng.* 48, 10 (2022), 4014–4040. <https://doi.org/10.1109/TSE.2021.3109563>
- [18] Huijuan Zhu, Yang Li, Ruidong Li, Jianqiang Li, Zhuhong You, and Houbing Song. 2021. SEDMDroid: An Enhanced Stacking Ensemble Framework for Android Malware Detection. *IEEE Trans. Netw. Sci. Eng.* 8, 2 (2021), 984–994. <https://doi.org/10.1109/TNSE.2020.2996379>

Received 2024-07-05; accepted 2024-07-26