Androlic: An Extensible Flow, Context, Object, Field, and Path-Sensitive Static Analysis Framework for Android*

Linjie Pan
State Key Lab. of Computer Science
Institute of Software, CAS
Univ. of Chinese Academy of Sciences
Beijing, China
panlj@ios.ac.cn

Baoquan Cui
State Key Lab. of Computer Science
School of Software and
Microelectronics, PKU
Beijing, China
cbq@pku.edu.cn

Jiwei Yan
Tech. Center of Softw. Eng
Institute of Software, CAS
Beijing, China
yanjw@ios.ac.cn

Xutong Ma
State Key Lab. of Computer Science
Institute of Software, CAS
Univ. of Chinese Academy of Sciences
Beijing, China
maxt@ios.ac.cn

Jun Yan†
State Key Lab. of Computer Science
Institute of Software, CAS
Univ. of Chinese Academy of Sciences
Beijing, China
yanjun@ios.ac.cn

Jian Zhang†
State Key Lab. of Computer Science
Institute of Software, CAS
Univ. of Chinese Academy of Sciences
Beijing, China
zj@ios.ac.cn

ABSTRACT

Static analysis is widely used to detect potential defects in apps. Existing analysis tools focus on specific problems and vary in supported sensitivity, which make them difficult to reuse and extend for new analysis tasks. This paper presents Androlic, a precise static analysis framework for Android which is flow, context, object, field and path-sensitive. Through configuration items and APIs provided by Androlic, developers can easily extend it to perform custom analysis tasks. Evaluation on an example program and 20 real-world apps show that Androlic can analyze apps with high precision and efficiency.

CCS CONCEPTS

• **Theory of computation → Program analysis**.

KEYWORDS

Static Analysis, Android, Sensitivity, Extensible

ACM Reference Format:
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 '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3293882.3339001

*This work is supported by National Natural Science Foundation of China (Grant No. 61672505), and Key Research Program of Frontier Sciences, CAS, Grant No. QYZDJ-SSW-JSC036.

†Corresponding Author.

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3339001

1 INTRODUCTION

Android is a popular mobile operating system, based on which many applications are developed. In order to detect problems in Android apps, the research community has developed many static analysis tools. These tools are developed for different purposes and thus incorporate various techniques with different precision. As we know, the precision of static analysis depends heavily on which sensitivities the tools takes into account. Without loss of generality, common sensitivities include flow, path and context sensitivity. For object oriented languages such as Java, we need to consider object and field sensitivity in addition. Theoretically, the more sensitivities are considered, the higher precision the analysis can reach.

Li et al. [10] summarized the number of sensitivities (flow, path, context, object, field) that popular Android static analysis tools considered. The result showed that most tools support a few of the five sensitivities and the number of sensitivities supported by these tools varies. Hopper [4] and Thresher [3] are the only two tools that support five sensitivities while they integrate sensitivity into the algorithm designed for concrete analysis tasks, which make them difficult to extend.

In view of this, we developed Androlic, a static analysis framework for Android which considers flow, context, path, object and field sensitivity. Our framework is built on top of Soot [9] and Jimple [13]. For each `SootMethod` under analysis, it carries out symbolic execution along its CFG. During symbolic execution, infeasible paths are eliminated and the call graph is built on-the-fly. In order to obtain precise call graphs, we build a heap model to process polymorphism and thus realize object and field sensitivity. Note that Androlic maintains complete points-to information and change points-to relation when strong update occurs. For statements containing method invocation, we check whether the invoked method is a library method. For a library method, we build a dummy object according to its return type and class hierarchy relationship supplied by Soot. For a non-library method, Androlic builds its concrete context and carries out context-sensitive inter-procedural analysis.

Besides sensitivity, Androlic provides flexible configuration items and abundant APIs so that developers can easily extend it to accomplish their own analysis tasks.

2 ANDROLIC

Androlic takes an apk file as input and performs symbolic execution based on CFG generated by Soot. Figure 1 shows the architecture of Androlic, which contains three modules as following:

- **Preprocessing.** Androlic takes an apk file as input, constructs the intra-procedural CFG and generates class hierarchy relationship via Soot. Considering the lifecycle of Android component, we build a dummy main method for each Activity as FlowDroid [2] does.
- **Object Oriented Modeling.** The object and field sensitivity are highly related to the characteristics of object orientation. On one hand, Androlid constructs store-based heap model through allocation sites [8]. On the other hand, Androlic takes *this* reference into consideration, building an extensive method context.
- **Symbolic Execution.** The symbolic execution engine of Androlic processes the Jimple statement along the intra-procedural CFG. During the process, infeasible paths are eliminated. And class initialization, which is always ignored [5], is taken into consideration.

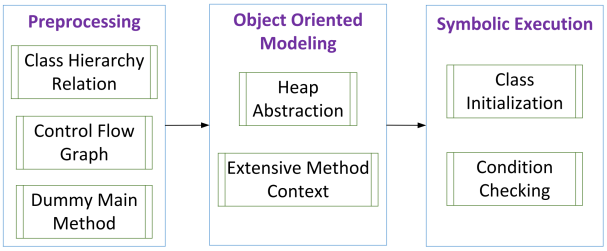

Figure 1: Architecture of Androlic

2.1 Object Oriented Modeling

The characteristics of object orientation is a vital factor that influences the precision of Android static analysis. In fact, object orientation is not only related to object and field sensitivity, but also affects the construction of call graph and thus the precision of inter-procedural analysis.

Heap Abstraction. The heap model in Androlic is store-based. We maintain a map from reference variables to allocation sites. In Jimple, the mapping relation can only be initiated in an `AssignStmt` where the left operand represents reference variable and the right one denotes allocation site.

There are two types of allocation sites in Androlic, i.e., explicit and implicit. According to the grammar of Jimple, we take three types of expressions, i.e., `NewExpr`, `NewArrayExpr` and `NewMultiArrayExpr` as explicit allocation sites where we can clearly assure the type of newly created objects. The `InvokeExpr` of library method is taken as the implicit allocation site. It is called 'implicit' since we can not obtain the body of library method and thus can not infer the concrete type of objects if the return type of the library method has subtypes. For implicit allocation sites, we randomly appoint the type of newly created object from the possible types deduced through class hierarchy relationship.

There are four types of reference variables in Jimple, i.e., `ArrayRef`, `StaticFieldRef`, `InstanceFieldRef` and `Local`. For a local variable or `StaticFieldRef`, we simply build a mapping relation from it to the allocation sites denoted by the right operand: $Var \rightarrow \mathcal{H}$. For `ArrayRef` and `InstanceFieldRef`, we define them as a tuple $\langle base, op \rangle$. In `ArrayRef`, *base* denotes the name of an array and *op* denotes the index. In `InstanceFieldRef`, *base* is the variable that holds a field and *op* is the field. The mapping relation is denoted as: $\mathcal{H} \times op \rightarrow \mathcal{H}$. The first $\mathcal{H}$ denotes the allocation site where *base* points to, and the second $\mathcal{H}$ denotes the allocation site of the right operand.

Extensive Method Context. In Java, the invocation of non static method is in the form of $var.methodName(parameter)$ where *var* is a reference variable pointing to an allocation site. In the invoked method, *var* is denoted by *this* reference. Considering the characteristics of encapsulation, the invoked method could manipulate the field of *var*. If we can not determine the heap object which *var* points to, the field sensitivity can not be guaranteed. Moreover, the type of heap object which *var* points to decides which method will be invoked at the call site [1]. That is to say, the precision of call graph construction highly depends on how we process *var*.

Therefore, when dealing with invocation of non static method, we not only replace the formal parameters with actual parameters, but also replace formal *this* with actual allocation site to build an extensive context for the invoked method. In other words, *var* is taken as a special formal parameter of the invoked method. We can easily replace *var* with the heap object it points to since Androlic maintains a mapping from reference variables to allocation sites as mentioned above.

2.2 Symbolic Execution

The symbolic execution engine processes Jimple statements along the CFG of input method (dummy main method is the input method by default), calculating symbolic value of base type variables and updating the mapping from reference variables to allocation sites. For statements containing invocation of non-library method, we save the status of current method and switch into the invoked method with extensive context introduced in section 2.1.

Class Initialization. The Java compiler encapsulates static code block and non final static field of a class into a special method called *clinit*, which will be invoked when the class is initialized. According to the semantics of Java, a class will be initialized whenever its object is first instantiated or its static method/field is used. Besides, the parent class must be initialized before the initialization of its subclasses. During symbolic execution, Androlic checks whether the class appearing in current statement is initialized and invokes *clinit* if the class has not been loaded yet.

Condition Checking. For a condition statement, Androlic first calculates the value of variables contained in the statement. If all variables correspond to concrete values, we can decide the satisfiability of the condition immediately and thus remove infeasible

paths. Here, concrete value includes numeric constant, string constant, *null* or explicit allocation site. Otherwise, Androlic takes all successive statements as feasible statements and saves the symbolic value of variables for potential extensibility.

2.3 Usage

Users can perform custom static analysis tasks through setting up configuration items and extending APIs provided by Androlic. More information about the usage of Androlic and the latest version can be found at github[1].

Configuration. As we know, path-sensitivity can easily raise path-explosion when the scale of program grows. In view of this, Androlic provides configuration items to limit the maximum path number during symbolic execution. For the same purpose, there are also configuration items to limit the maximum number of loop unrolling and the maximum level of recursive invocation. Users can also set the maximum running time of analyzing an apk. Moreover, users can also appoint a method instead of the dummy main method as the input method of symbolic execution engine. Configuration items of Androlic and their default values are listed in Table 1.

Table 1: Configuration Items of Androlic

Configuration Item	Default
MaxPathNum	40000
MaxRecursionNum	0
MaxUnrollingNum	1
MaxRunningTime	30 minutes
EntryMethod	DummyMainMethod

Extensibility. By implementing APIs of Androlic, developers can perform different static analysis tasks according to concrete scenarios. Firstly, developers can define custom approaches to process library method invocation. As mentioned above, the invocation of a library method is taken as implicit allocation site in our heap model by default. By extending Androlic, users can replace implicit allocation sites with explicit ones to obtain more accurate analysis result. Secondly, users can add custom operations at each step of symbolic execution. The symbolic execution engine simulates the running process of program and users can instrument the engine to record specific information. Last but not least, Androlic maintains the symbolic expression of variables so that users can leverage constraint solver to further remove infeasible paths and generate test cases.

3 CASE STUDY

In this section we demonstrate the characteristics of Androlic through an example program.

In Listing 1, we define three classes: `Adult`, `University` and `Person`. Note that `Person` is a subclass of `Adult` and it contains a field `graduation` whose type is `University`.

Listing 1: Classes under Analysis

```
1  package com;
2  public class Adult {
3     public static int minAge = 18;
4  }
5  class University{
6     private String name;
7     public String getName() {
8        return name;
9     }
10    public University(String name) {
11       this.name = name;
12    }
13 }
14 class Person extends Adult {
15    private int age;
16    private University graduation;
17    public University getGraduation() {
18       return graduation;
19    }
20    public int getAge() {
21       return age;
22    }
23    public void setGraduation(University graduation) {
24       this.graduation = graduation;
25    }
26    public Person(University university, int theAge) {
27       this.graduation = university;
28       this.age = theAge;
29    }
30 }
```

In Listing 2, we define a method as the entry method for symbolic execution engine of Androlic. In the entry method, we first declare three `University` objects and two `Person` objects. In the following code, we make manipulation on these objects.

Listing 2: Entry Method of Symbolic Execution

```
1  public void entryMethod() {
2     University peking = new University("peking");
3     University tsinghua = new University("tsinghua");
4     University USTC = new University("USTC");
5     Person ming = new Person(peking, 21);
6     Person hong = new Person(tsinghua, 20);
7     if( Adult.minAge == 18 ) {
8        System.out.println("min age of adult is 18");
9     } else {
10       System.out.println("min age of adult is not 18");
11    }
12    if( ming.getAge() == hong.getAge() ) {
13       System.out.println("They have the same age");
14    } else {
15       System.out.println("They do not have the same age");
16       ming.setGraduation(USTC);
17       ming.setGraduation(tsinghua);
18       if( ming.getGraduation() == hong.getGraduation() )
19          System.out.println("Their graduate is the same");
20       else
21          System.out.println("Their graduate is different");
22    }
23 }
```

There are many non-library method invocations in Listing 2, each invocation will trigger inter-procedural context-sensitive analysis. We take line 5 of Listing 2 where *ming* is appointed to a `Person` object as example. Androlic invoked the constructor method of `Person` and its parent class `Adult`. Here, indent denotes that a new method is invoked:

```
1  specialinvoke $r3.<com.Person:void <init>(com.University,int)>($r4, 21)
2     $r0 := @this: com.Person
3     $r1 := @parameter0: com.University
4     $i0 := @parameter1: int
5     specialinvoke $r0.<com.Adult: void <init>()>()
6        $r0 := @this: com.Adult
7        specialinvoke $r0.<java.lang.Object: void <init>()>()
8        return
9     specialinvoke $r0.<com.Adult: void <init>()>()
10    $r0.<com.Person: com.University graduation> = $r1
11    $r0.<com.Person: int age> = $i0
12    return
13 specialinvoke $r3.<com.Person:void <init>(com.University,int)>($r4, 21)
```

[1]https://github.com/pangeneral/Androlic

There are three condition statements in the entry method which can generate 6 potential paths in total. As mentioned in section 2.2, Androlic invokes the `clinit` method of `Adult` and checks the satisfiability of each condition statement to generates the only feasible path:

```
1  $i0 = <com.Person: int minAge>
2  if $i0 != 18 goto $r6 = <java.lang.System: java.io.PrintStream out>
3  $r6 = <java.lang.System: java.io.PrintStream out>
4  virtualinvoke $r6.<java.io.PrintStream: void
       println(java.lang.String)>("min age of adult is 18")
5  $i0 = virtualinvoke $r3.<com.Person: int getAge()>()
6  $i1 = virtualinvoke $r2.<com.Person: int getAge()>()
7  if $i0 != $i1 goto $r6 = <java.lang.System: java.io.PrintStream out>
8  $r6 = <java.lang.System: java.io.PrintStream out>
9  virtualinvoke $r6.<java.io.PrintStream: void
       println(java.lang.String)>("They do not have the same age")
10 virtualinvoke $r3.<com.Person: void setGraduation(com.University)>($r1)
11 virtualinvoke $r3.<com.Person: void setGraduation(com.University)>($r5)
12 $r1 = virtualinvoke $r3.<com.Person: com.University getGraduation()>()
13 $r4 = virtualinvoke $r2.<com.Person: com.University getGraduation()>()
14 if $r1 != $r4 goto $r6 = <java.lang.System: java.io.PrintStream out>
15 $r6 = <java.lang.System: java.io.PrintStream out>
16 virtualinvoke $r6.<java.io.PrintStream: void
       println(java.lang.String)>("Their graduate is the same")
```

The printed message in line 4, 9 and 16 prove that Androlic processes method invocation and object/field correctly. In other words, Androlic is context, object and field-sensitive. Note that the statements of invoked method are omitted due to the limit of space.

4 EVALUATION

To evaluate the effectiveness of Androlic, we collect 20 real-world apps from F-Droid [6] and Wandoujia app market [14]. Androlic analyzes these apps under default configuration as Table 1 shows, which means the dummy main method of each Activity is taken as the entry method of symbolic execution engine.

Table 2 shows the result of experiment. The column *App* and *Size* denotes the name and size (KB) of apps under analysis respectively. The first ten apps are collected from F-Droid and the latter ten apps come from Wandoujia. The column *invalid* and *valid* denotes the number of analyzed Activities whose path number is equal to or greater than and less than 40000 (`MaxPathNum`) respectively. The column *Average* denotes the average number of paths of valid Activities. The column *Min* and *Max* denote the minimum and maximum number of paths of valid Activities respectively. The column *Time* denotes the analysis time (second) of each app. Apparently, the path num of most Activities is less than the threshold and Androlic can analyze these apps within 30 minutes (`MaxRunningTime`).

5 RELATED WORK

Symbolic execution is a static analysis technique which replaces concrete value with symbolic value to execute the program. Symbolic PathFinder (SPF) [12] is a popular static analysis tool for Java bytecode that combines symbolic execution with model checking. However, SPF did not consider the characteristics of Android such as lifecycle and entry-point. Recently, some researchers leveraged symbolic exeuction in Android testing [7, 11, 15].

Compared with previous work, Androlic achieves full sensitivity and shows strong extensibility which not only can be used in test case generation, but also other analysis tasks such as bug detection by adding self-defined operations into the symbolic engine. Moreover, Androlic takes lifecyle, callback-methods and entry-points into consideration, constructing dummy main method for Android analysis.

Table 2: Results of Symbolic Execution on Real-world Apps

App	Size	Activity		Path Information			Time
		invalid	valid	Average	Min	Max	
2048	859	0	1	9411	9411	9411	6
24game	2540	0	1	11	11	11	1
AAT	2327	0	18	3	3	3	1
ABCore	1205	0	9	269	7	1121	1
AcrylicPaint	451	0	4	9	4	11	1
ActivityDiary	3524	1	10	2978	6	16535	206
aGrep	344	0	6	4761	19	12803	12
APhotoMap	1406	0	12	838	5	9986	34
ATimeTracker	1309	2	3	57	5	163	24
webSearch	1898	0	3	5376	4013	6195	15
danshouhuahua	2389	1	55	9059	7	14711	133
googleearth	12785	0	12	3082	11	10000	124
gugepinyin	18136	1	24	1126	6	19889	29
jijianhuilv	12414	2	19	4196	7	9227	63
lijidai	12883	5	29	7372	4	32403	313
paizhaofanyi	9121	0	40	4986	7	10000	229
pingduoduo	18674	0	109	2794	11	6923	264
wenzisaomiao	13899	1	8	3267	11	7019	119
youdaobeidanci	15110	3	47	9337	3	14713	253
zhaopianhuifu	1954	2	21	831	7	7159	34

6 CONCLUSION

This paper presents a flow, context, object, field and path-sensitive static analysis framework, which considers the characteristics of object orientation. With flexible configuration items and abundant APIs, users can easily implement static analysis tasks under specific requirements. In the future, we will extend the framework to process more Java and Android library methods to achieve higher precision.

REFERENCES

[1] G. Agrawal. Demand-driven construction of call graphs. In *CC*, pages 125–140, 2000.
[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI*, pages 259–269, 2014.
[3] S. Blackshear, B. E. Chang, and M. Sridharan. Thresher: precise refutations for heap reachability. In *PLDI*, pages 275–286, 2013.
[4] S. Blackshear, B. E. Chang, and M. Sridharan. Selective control-flow abstraction via jumping. In *OOPSLA*, pages 163–182, 2015.
[5] M. Christakis and C. Bird. What developers want and need from program analysis: an empirical study. In *ASE*, pages 332–343, 2016.
[6] F-Droid. https://f-droid.org.
[7] X. Gao, S. H. Tan, Z. Dong, and A. Roychoudhury. Android testing via synthetic symbolic execution. In *ASE*, pages 419–429, 2018.
[8] V. Kanvar and U. P. Khedker. Heap abstractions for static analysis. *ACM Comput. Surv.*, 49(2):29:1–29:47, 2016.
[9] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *CETUS*, volume 15, page 35, 2011.
[10] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. L. Traon. Static analysis of Android apps: A systematic literature review. *Information & Software Technology*, 88:67–95, 2017.
[11] N. Mirzaei, S. Malek, C. S. Pasareanu, N. Esfahani, and R. Mahmood. Testing Android apps through symbolic execution. *Software Engineering Notes*, 37(6):1–5, 2012.
[12] C. S. Pasareanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for Java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013.
[13] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying Java Bytecode for Analyses and Transformations, 1998.
[14] Wandoujia. https://www.wandoujia.com.
[15] C. Yeh, H. Lu, C. Chen, K. K. Khor, and S. Huang. CRAXDroid: Automatic Android system testing by selective symbolic execution. In *SERE-Companion Volume*, pages 140–148, 2014.