A Model Guided Security Analysis Approach for Android Applications

Yan Zhang[1, 2*], Zhoujun Li[1], Dianfu Ma[1]

[1] School of Computer Science and Engineering, Beihang University, Beijing, China.
[2] School of Mathematics and Computer Science, Hubei University, Wuhan, China.

* Corresponding author. Tel.: +86-10- 82338247; email: zttyan@sina.com
Manuscript submitted February 20, 2016; accepted May 8, 2016.
doi: 10.17706/jsw.11.7.677-684

Abstract: Revealing security vulnerabilities is one of great challenges for the Android ecosystem. Static analysis is the usual approach of the security analysis for computer software. However, it is undirected and time-consuming for the common static analysis methods to analyze the entire Android application systematically from the main entry point. In order to adapt to the event-driven feature of Android applications, a model guided security analysis approach for Android applications is introduced and implemented into the prototype tool MSAS. This approach builds and utilizes the Operation Security Model to guide the targeted analysis process, and then concentrate on the identified analysis surface to reduce analysis workload, thereby achieving fast analysis speed and on-demand code coverage based on the security rules. The test result shows that this approach can improve the efficiency and effect of security analysis for Android applications, and it has revealed 11 security vulnerabilities by analyzing several popular Android applications.

Key words: Model guided analysis, security analysis, Android application security, static analysis, vulnerability discovery.

1. Introduction

Android [1] is a popular open source operating system for mobile and embedded devices, and millions of kinds of Android applications are nowadays running on a variety of smart devices including mobile phones, pads, tablets and smart televisions. The Android applications have been influencing people's lives and global economy so deeply that their computational errors and security vulnerabilities can cause financial ruin, privacy leaks as well as heavy loss to information asset. Therefore, security analysis is important for the Android developers and security auditors to discover and locate the unknown bugs and vulnerabilities in Android applications to prevent the potential calamities.

The static analysis method [2] is often used in the security analysis for computer software. Great efforts have been devoted to the development of static analyzers. The analyzers can perform on the source code as well as the byte code, and can implement the solutions for the program states without actually executing programs. Because the source code of an Android application contains lots of Java code, many static analysis tools for Java source code can serve as references [3], such as Checkstyle, PMD, FindBugs, Fortify SCA, Klocwork, Coverity [4].

The Android application will be analyzed by unpacking the target APK package, analyzing the optimized Dalvik bytecode [5] and detecting security vulnerabilities. The following features of the Android platform need to be considered for better quality of the security analysis. First, the Android applications are developed by Java language with the extended Android libraries and components for the interactive operations

and mobile functions. Next, the Android application model is based on an event-driven architecture [6], which supports data sharing and function interoperating across different applications. For adapting to these features, a model guided analysis approach is proposed to improve the efficiency and effect of security analysis for Android applications.

The rest of this paper is organized as follows. Section 2 describes in detail the analysis process as well as the relevant concepts. Section 3 presents the experimental results on several popular Android applications, based on our approach implementation. Section 4 is to make conclusion of the paper.

2. Analysis Approach

The general framework of the model guided security analysis approach is depicted on Fig. 1. In the approach process, the APK package of the target Android application are firstly unpacked to extract the application crucial files such as AndroidManifest.xml and classes.dex. Next, the application interface analysis will be performed on AndroidManifest.xml to locate the exposed interfaces and the input data structures of the application components. The potentially malicious data can be received by data sharing and function interoperating across different applications. Then, we can retarget and disassemble classes.dex to obtain the Dalvik bytecode, and then perform the fundamental structure analysis including control flow analysis [7] for the construction of the class hierarchy and the control flow graph.

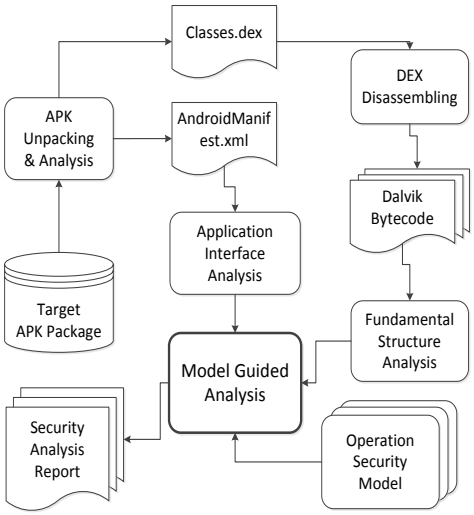

Fig. 1. General framework of the model guided security analysis for Android applications.

After that, the model-guided analysis can begin from application interface points to traverse the control structures of the relevant methods. During the traversal, it uses the Operation Security Model to track the sequence of the important application operations and guide the targeted analysis in order to achieve on-demand security analysis. The security violation of the application operations can be recorded and analyzed to detect security vulnerabilities or bugs and generate the security analysis report. The following sections will discuss the key procedures in detail.

2.1. Build the Operation Security Model

In order to reduce the number of the potential security vulnerabilities, the important operations of an Android application have to comply with the specific security rules. The security rules can be described in the form of the OSM model i.e. Operation Security Model.

The OSM model is based on the finite state machine, which is the mathematic model with sound theoretical foundation [[8]]. This model can have discrete inputs and outputs, and use the states to keep record of the effect of the past behaviors of the Android application. The formal notation of the OSM model is as follows:

$$\text{OSM} = (S, O, \Sigma, \delta, s_0, T, V) \quad (1)$$

This notation involves the following seven components. S denotes the finite set of the application states, where $s_0 \in S$ denotes the initial state in the application entrance.

O denotes the finite set of the application operations, including library function calls e.g. android.content.IntentFilter.addAction. Σ denotes the finite set of the operation primitives, where an operation primitive consists of a series of application operations in the predefined sequence. $\delta: S \times \Sigma \to S$ denotes the state transition function which gives the target state transited after performing an operation primitive. T denotes the state set of application normal terminations. V denotes the set of application vulnerable states, which means that the previous operation sequences drive the application into some potential dangerous state.

Fig. 2 shows an example of the typical OSM for detecting the exposed and unprotected components. The definition of the OSM states and transitions is shown respectively in Table 1 and Table 2. In the Android application, the inter-application communication can be received and sent in the form of the intent messages for the application cooperation and data sharing. The intent filters can be registered dynamically on demand by the application components, such as activities, services, etc. However, the component which defines the intent filter will made itself public and exposed to other applications that can send the correct intent message. The exposed components may become potential vulnerable if some relative library functions are called in the specific sequence. That can be detected by the following OSM.

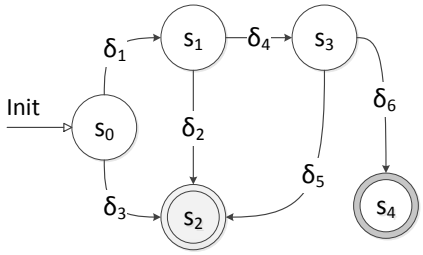

Fig. 2. OSM for detecting the exposed and unprotected components.

In this model, the intent states are tracked and checked. An intent message can be in implicit intent state by the method calls to several variants of android.content.Intent.init with no specified reactive component or class. Next, the intent state can get secure from implicit to target by calling the method Intent.setComponent or Intent.setClass to actively attach the target compontents to the intent message. Then, as the intent detailed arguments, the extra data can be attached to the intent message by the method Intent.putExtra.

However, if the implicit intent message with extra data is sent unprotected by the methods such as android.content.Context.sendBroadcast passing null as permission, the vulnerable state is reached. It may lead to the potential harmfulness, including unintended leakage of sensitive information, unauthorized execution of component functionality, etc.

The OSM model can imposes security rules and constraints on the operation sequences executed by the target Android application. It can also give the heuristic guidance information by $\delta: S \times \Sigma \to S$ for the targeted analysis in the relevant control structures.

Table 1. State Set of the Example OSM

State	State Description
s_0	Initial State
s_1	Implicit Intent State
s_2	Target Intent State
s_3	Extra Data Attached State
s_4	Vulnerable State

Table 2. State Transitions of the Example OSM

Transition	Effect Description
δ_1	Create Implicit Intent Message
δ_2	Specify Target Compontent in Intent Message
δ_3	Create Target Intent Message
δ_4	Attach Extra Data to Intent Message
δ_5	Attach Target Compontent to Intent Message
δ_6	Send Unprotected Intent Message

2.2. Locate the Analysis Surface

The code analysis based on the OSM model will begin traversing the control structures of the relevant methods from application interface points, which can be called the analysis surface.

The common static analyzers are usually designed to begin the whole analysis process from the program main method such as int main(int argc, char *argv[]) in the C/C++ programs. However, the Android application model is based on an event-driven architecture [6], thus has more than one single entry point. There are several call interfaces, which can be considered as input event handlers. When some specific input event occurs, many, the Android application will call the responding method on demand for the event handlers.

The target Android application is firstly unpacked from the APK package to extract the application crucial files such as AndroidManifest.xml and classes.dex. These call interfaces can be located by analyzing the formal description of application components and the relative method reflection mechanism [5], according to the manifest file AndroidManifest.xml. This manifest file can register several types of the application components in the XML form: (1) activities, which render the visual user interfaces and interact with the user operations; (2) services, which have no interaction and perform the longtime background operations; (3) broadcast receivers, which receive and handle the broadcast messages and (4) content providers, which provide the controlled data sharing from the inner data sources including databases and files.

The call interface example of the exposed application components is depicted in the following manifest snippets extracted from the OWASP project “Fourgoats”.

```
<activity android:name=".activities.ViewCheckin" android:exported="true" />
<receiver android:label="Send SMS"
android:name=".broadcastreceivers.SendSMSNowReceiver">
    <intent-filter>
        <action
android:name=".fourgoats.SOCIAL_SMS" />
    </intent-filter>
</receiver>
<service
android:name=".services.LocationService">
     <intent-filter>
```

```
        <action
android:name=".fourgoats.services.LocationService" />
      </intent-filter>
</service>
```

Next, the call interfaces will be linked to the relative methods in the Dalvik bytecode of the executable file classes.dex in the Android applications. These Android bytecode can be retargeted from Dalvik to Java bytecode by the translator tools including Dex2jar, Undx and Dare [[9]]. Then, using the Java bytecode tools such as FindBugs [[10]], the target bytecode can be disassembled and processed to accomplish the fundamental structure analysis, including control flow analysis for the construction of the class hierarchy and the control flow graph. Our analyzer will find the above call interface by following their class names described in AndroidManifest from the class hierarchy. Then, it chooses the responding method by Java reflection, based on the type and feature of the application component. These methods will constitute the analysis surface as the entry points in the whole process of detecting security vulnerabilities.

2.3. Perform the Model Guided Analysis

The model-guided analysis utilizes the pre-constructed Operation Security Model and the analysis surface to perform the targeted analysis for the Android application. The advantage of this analysis is that it can achieve fast analysis speed and on-demand code coverage based on the upcoming examined security rules.

Algorithm 1
ALGORITHM FOR THE MODEL GUIDED ANALYSIS APPROACH

Input: Definition of the OSM and the analysis surface
Output: Security analysis report of the Android application

```
1:  for each method M in the analysis surface do
2:    O_M ← operations performed in M
3:    for each model D in the OSM do
4:      if O_M ∩ D.O = ∅ then
5:      continue to next model
6:      end if
7:    S_M ← M.s0while select transition path p from S_M to
8:      M.V do
9:        σ_P ← {Operation o|o ∈ p, p = (δ_i, … , δ_j)}
10: while select execution trace E from M.CFG
11:      with the current most set {σ_P ∩ E} do
12:          while E follows p do
13:            if E encounts M.V then
14:          pc ← path conditions collected in E
15:           solution = ConstraintSolve(pc)
16:              if solution ≠ ∅ then
17:                Report. add(E, p, solution)
18:                break to another model D
19:              end if
20:            end if
21:            if E encounts M.T then
22:              break while
23:            end if
24:          end while
25:        end while
26:      end while
27:    end for
28: end for
```

Algorithm 1 provides the key points of the model guided analysis approach. The algorithm starts with the OSM and the analysis surface as input. First, we extract the relevant methods of the analysis surface located in the above step and start traversing the control constructions from one of these call interfaces (lines 1–2). Then, The OSM model are selected one by one for security checking (line 3). After the unrelated method is beforehand found and skipped (lines 4–6), the targeted analysis start to work (lines 7–28). The transition paths leading to security violation are enumerated and utilized to guide the selection of the execution trace. It speeds up the security analysis by direct coverage of the suspicious code regions based on the security constraints imposed by the OSM (lines 7–11). The OSM tracks the suspicious execution traces of the important application operations and checks them with the security constraints (lines 12–23). However, not all the execution traces that follow the transition paths ending at the application vulnerable states are feasible. Therefore, the path conditions in the execution traces are collected and converted to the SMT (Satisfiability Modulo Theory) problem, then they are solved by calling the constraint solver [[11]] (lines 13–15). The execution traces with no solution are pruned in order to decrease the false positives. On the contrary, the execution traces with path solutions are feasible, meaning that the security violation of the application operations will occur. And then the path and state transitions are recorded in the security analysis report as security vulnerabilities or bugs (lines 16–19).

In the model-guided analysis, the targeted analysis is designed to adapt the features of Android application. The Android system supports the implicit intent messages to call the application components from the other external applications. These external applications may be controlled by potentially malicious users and intend to inject the attack data directly without user intervention. The attack data can spread out by the involvement of assignments and computations, after they enter the application from the call interfaces. If we employ the common static analysis method to analyze the entire Android application systematically from the main entry point, it will take long time to reach the aforementioned dangerous code regions, even to result in missing them. The model-guided analysis differs in direct coverage of the analysis surface by exercising these exposed and callable interface code comprehensively. It also differs in the guidance for the path exploration because that the model OSM can not only track the dangerous operations and vulnerable states, but also involve the purification operations to mitigate or remove the harmful effect of the earlier operations. Thus, it can achieve fast analysis speed and on-demand code coverage by target analyzing the suspected attack path based on the model guidance.

3. Implementation and Experiment

The approach presented above has been implemented into MSAS (the Model-guided Security Analysis System). We have designed and developed MSAS using Java language.

MSAS utilizes the toolkit APKTOOL to unpack the APK package to extract the application crucial files including the decoded manifest file AndroidManifest.xml and the Dalvik bytecode package of classes.dex. It then calls the toolkit Dex2jar to retarget classes.dex from Dalvik bytecode to Java bytecode [9]. Next, the core engine of MSAS can analyze the Java bytecode to accomplish the analysis surface identification and the fundamental structure analysis, based on FindBugs [10], an open source static analysis tool for Java. For this work, we extend FindBugs by target exploration for the relevant paths under the guidance of the model OSM. For the model construction, MSAS uses the XML/SCXML language [[12]]to describe the states, transitions and operations of the model and provides a friendly workbench to model the operation specification graphically into the form of FSM.

The prototype tool of MSAS has been developed and applied to analyze several popular Android applications including Huawei Dbank, Baidu Netdisk, Xiaomi Miliao, Snda Youni, Renren Mobile, etc. We have con-

ducted these experiments on a Thinkpad laptop (Intel Core i7 2.7 GHz quad-core with 4GB RAM), running Windows XP SP3.

Table 3. Experiment Results of MSAS

App	Size(MB)	#Com	#Target	#Vul	Time(s)
Huawei	2.32	33	24	2	47.1
Baidu	7.35	43	22	1	69.5
Xiaomi	14.1	194	51	3	227.6
Snda	13.7	34	21	3	94.3
Renren	16.9	68	34	2	125.8

(a) #Com – the component number of the Android application.
(b) #Target – the component number of the targeted analysis.
(c) # Vul – the number of the detected vulnerabilities.

Table 3 shows the experiment results of MSAS. The Operation Security Model guides the targeted analysis process and concentrate on the analysis surface to reduce the analyzed interface methods by even more than 70% (from 194 to 51). Therefore, compared with the common static analysis approaches, MSAS can achieve fast analysis speed. For example, it takes no more than 100 seconds to accomplish the security analysis for the Android application of about 10MB. During these experiments, MSAS has revealed 11 security vulnerabilities, including 8 vulnerabilities of unauthorized access for Xiaomi Miliao, Snda Youni, Renren Mobile, 3 vulnerabilities of authentication bypass for Huawei Dbank as well as 1 vulnerability of denial of service for Baidu Netdisk.

4. Conclusion

In this paper, a model guided security analysis approach for Android applications is introduced and implemented into the prototype tool MSAS. The approach builds and utilizes the Operation Security Model to guide the targeted analysis process in order to achieve fast analysis speed and on-demand code coverage. The test result shows that it can improve the efficiency and effect of security analysis for Android applications.

This work was supported by the National Natural Science Foundation of China (Grant No. 61170189, 61202239, 61300172).

References

[1] Shabtai, A., Fledel, Y., Kanonov, U., *et al*. (2010). Google Android: A comprehensive security assessment. *IEEE Security and Privacy*, *8*, 35-44

[2] Wichmann, B. A., Canning, A. A., Clutterbuck, D. L., *et al*. (1995). Industrial perspective on static analysis. *Software Engineering Journal*, *3*, 69–75

[3] Livshits, V. B., & Lam, M. S. (2005). Finding security vulnerabilities in Java applications with static analysis. *Proceedings of the 14th conference on USENIX Security Symposium on Association*.

[4] Ware, M. S., & Fox, C. J. (2008). Securing Java code: Heuristics and an evaluation of static analysis tools. *Proceedings of the 2008 workshop on Static Analysis*.

[5] Liu, J., & Yu, J. (2011). Research on development of android applications. *Proceedings of 4th International Conference on Intelligent Networks and Intelligent Systems. Kunming*.

[6] Butler, M. (2011). Android: Changing the mobile landscape. *IEEE Pervasive Computing*, *10*, 4-7

[7] Nielson, F, Nielson, H. R., & Hankin, C. (1999). *Principles of PROGRAM analysis*. Springer Publishing.

[8] Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation* (3rd Edition). Addison-Wesley Longman Publishing Inc, Boston, USA.

[9] Octeau, D., Jha, S., & McDaniel, P. (2012). Retargeting Android applications to Java bytecode. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*.
[10] Ayewah, N., Hovemeyer, D., Morgenthaler, J. D., *et al*. Using static analysis to find bugs. *IEEE Software*.
[11] Moura, L, & Bjørner, N. (2008). An efficient SMT solver. *Proceedings* of t*he 14th conf on Tools and Algorithms for Construction and Analysis of Systems*.
[12] Barnett, J., Akolkar, R., Auburn, R. J., *et al*. State chart XML (SCXML): State machine notation for control abstraction. *W3C Candidate Recommendation*.

Yang Zhang received the B.S. and M.S. degrees from School of Mathematics and Compuer Science at Hubei University, China in 1997 and 2000 respectively. Now he is a Ph.D. cadidate in the School of Computer Science and Engineering at Beihang University, and an associate professor in School of Mathematics and Computer Science at Hubei Univesity. His research interests include computer architecture, information system, and system security.

Zhoujun Li received the B.S. in computer science from Wuhan University, China in 1984, and the M.S. and the Ph.D. in computer science from National University of Defense Technology, China in 1986 and 1999 respectively. He was a professor in the Department of Computer Science at National University of Defense Technology from 1986 to 2005. Now he is currently a professor in the School of Computer Science and Engineering at Beihang University. His research interests include computer architecture, formal verification, and security protocol analysis.

Dianfu Ma received the B.S., M.S. and Ph.D. in computer science from Beihang University, China in 1982, 1985 and 1990 respectively. Now he is currently a professor in the School of Computer Science and Engineering at BeiHang University. His research interests include service computing, formal verification, and real-time system.