FlowSlicer
A Hybrid Approach to Detect and Avoid Sensitive Information Leaks in Android Applications using Program Slicing and Instrumentation

Luis Menezes and Roland Wismüller
Institute of Computer Science, Universität Siegen, Siegen, Germany

Keywords: Static Analysis, Dynamic Analysis, Hybrid Analysis, Slicing, Android, Privacy, Information-flow.

Abstract: With the increasingly amount of private information stored in mobile devices, the need for more secure ways to detect, control and avoid malicious behaviors has become higher. The too coarse-grained permission system implemented in the Android platform does not cover problems regarding the flow of the data acquired by the apps. In order to enhance detection, awareness and avoidance of such unwanted information flows, we propose a hybrid information flow analysis that mixes the benefits of static and dynamic analysis, using slicing and instrumentation. Our results indicate a precise detection and only a small overhead while running the application. The validation of our method has been done by creating a tool called FLOWSLICER and using the category *AndroidSpecific* from the DROIDBENCH repository of applications with known information leaks.

1 INTRODUCTION

Mobile devices, such as smartphones and tablets are important devices we all use in our daily life. The capability of providing tools and storing a big amount of data that we daily need made them become a truly indispensable device. Much of this success is only possible because of the ability to conveniently download applications directly from some repositories. The Android platform has several hundred thousands of applications divided in different markets (Arp et al., 2014), including an official one, the Google Play, and is worldwide known as one of the most popular platforms used in mobile devices. However, along with all benefits, there is still margin for malicious developers who try to use the coverage, power of storing and processing of such platforms to pose a threat to the user's privacy, security and also safety.

Modern operating systems, such as Android, use a permission system in which each application has a different set of permissions based on its requirements. It uses as a premise the fact that every application will only ask for the set of permissions that they need (Felt et al., 2011b). However, one of the problems of this approach is the lack of awareness from the user side (Felt et al., 2011a). The permissions are normally too coarse-grained, which means that a permission such as `android.permission.INTERNET` may be used either for retrieving server content or sending private data to a server (Jeon et al., 2012). Another problem in this approach is the fact that it only takes care of the access control of the requested data, i.e., granting or denying the access to a specific resource. Once the access is given, no further control is used to make sure the information is used properly (Hedin and Sabelfeld, 2011). However, in order to acknowledge what happens to the information one just gave access to, an information-flow control approach needs to be used. Information-flow control tracks the propagation of the information through the program and aims to make sure that the program handles the information securely (Hedin and Sabelfeld, 2011).

To address this issue, most of the existing solutions are using either static or dynamic analysis. Static analysis works by scanning the application's code to examine all possible execution paths and variable values without executing the application, making the process of analysis fast, repeatable and not dependent on the application execution (Kulenovic and Donko, 2014). In other hand, it counts with the downside of reporting false positives and also false negatives (Walden and Doyle, 2012). False positives occur because static analysis can only determine *that* an expression, e.g., a branch condition, depends on another one, but usually not *how exactly* it depends on the other. Thus, static analysis may consider execution paths as feasible which actually can never occur in reality. On the other hand, false negatives may happen

Menezes, L. and Wismüller, R.
FlowSlicer - A Hybrid Approach to Detect and Avoid Sensitive Information Leaks in Android Applications using Program Slicing and Instrumentation.
DOI: 10.5220/0006428604500455
In *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) - Volume 4: SECRYPT*, pages 450-455
ISBN: 978-989-758-259-2

because of inaccuracies in the analysis' abstraction. Dynamic analysis is performed while the application is actually executed, thus it is not susceptible to false positives. However, dynamic analysis only checks a concrete program execution, thus, it can never provide a proof that an application does not leak sensitive information. In addition, this approach may have a large negative impact on the application's performance.

In order to keep user's data private, we propose FLOWSLICER, a hybrid approach to detect sensitive information leaks in Android applications by mixing the benefits of static and dynamic analysis.

In this paper, we examine the effects of our approach by testing well-known malicious applications present in the ANDROIDSPECIFIC category from the DROIDBENCH repository in order to analyze both the precision of FLOWSLICER and its runtime overhead. The contributions of this work are: a hybrid method to detect information leaks on Android devices.

The rest of this paper is organized as follows: The state-of-art and some related work that inspired our method are presented in Section 2. FLOWSLICER is presented in Section 3. The evaluation of the method is presented in Section 4. Section 5 brings the conclusion of the work.

2 RELATED WORK

2.1 Static Analysis Approaches

DROIDSAFE (Gordon et al., 2015) presents a system for analyzing explicit flows of sensitive information in Android applications by performing an information-flow analysis based on a model obtained from the Android Open Source Project (AOSP) implementation. The information-flow analysis is performed by using an object sensitive points-to-analysis, to cover code that heavily uses object-oriented language features (such as inheritance and polymorphic code reuse), and flow insensitivity, considering all possible runtime event orderings that asynchronous callbacks can trigger. FLOWDROID (Arzt et al., 2014) was for a long time the state-of-the-art in the field of static analysis for Android applications. It implements a precise model of Android's lifecycle which allows the analysis to properly handle callbacks invoked by the Android framework, while context-, flow-, field- and object-sensitivity reduces the number of false alarms. For evaluation, FLOWDROID uses SecuriBench Micro (Livshits, 2005), which is a benchmark set designed for web applications, DROIDBENCH and a set of well-known Android test applications. It achieves 93% recall and 86% precision. (Fan and Xuan, 2016) propose a model of a dependence-based taint analysis for web applications. It uses the concept of source and sink methods in order to detect if there is a flow of information coming from the former to the latter.

Although the current static analysis frameworks provide valuable categorization for different Android applications, there are still several limitations in the analysis, since the static analysis can only perform an approximation of the real behavior of the application under focus. It assumes that some of the flow paths may be used while the application is running. However, it is not asured to happen, since some of the paths may depend on runtime variables.

2.2 Dynamic Analysis Approaches

TAINTDROID (Enck et al., 2010) is an information-flow tracking system for monitoring privacy in real-time on Android smartphones. It incurs 14% performance overhead on a CPU-bound microbenchmark with low overhead when running thirdparty applications. It works by categorizing privacy-sensitive data sources and labels when applications obtain information from these sources. It also performs system-wide tracking of variables, files and interprocess messages that propagate these data. DROIDBOX (Lantz, 2011) modified the Android's core libraries in order to employ an integrated system, containing TAINTDROID.

Although performing a more accurate analysis, the presented approaches still have some problems such as the incurrence of overhead that may impact the performance of the analyzed application and also the need of modifying the system, which may forbid normal users to use this approach, since it needs more knowledge around the Android modification field in order to install the system.

3 METHODOLOGY

FLOWSLICER is an approach that mixes a conservative static analysis with a dynamic analysis using a tagging architecture. The static analysis is able to identify the important information flow and, then, filter important parts of the flow to be analyzed with a dynamic analysis, that makes use of a tagging architecture to keep track of all sensitive objects that appears on the filtered flow. The hybrid approach presented here results in a high detection precision, small overhead and no false positives.

3.1 Input

FLOWSLICER takes as input a list of source and sink methods and the APK file to be analyzed. The list of source and sink methods is provided by SUSI (Arzt et al., 2013), a tool that classifies the methods present in the Android API as source, sink or neither, with a recall and precision of more than 92%, as evaluated using ten-fold cross validation. The APK file is analyzed by extending the Soot Framework (Vallée-Rai et al., 1999), which provides important resources for a precise analysis.

3.2 Entry Points

Since Android applications, unlike other Java programs, do not have an unique main method (Arzt et al., 2014), different components must be analyzed in order to create a main method and run the analysis. FLOWSLICER uses the *AndroidManifest.xml* file of each application under analysis to, initially, detect which components are used and, afterwards, to create the main method taking the extracted information into account.

The components present in the Android platform are: *Activity*, *Service*, *ContentProvider* and *BroadcastReceiver*. *Application* is also provided by the *AndroidManifest.xml* file and works as a base class to handle the data of the whole application. In order to obtain the correct flow of data within the application, the lifecycle of each component and also the *Application* class need to be taken into account, as well as the order they may be called. FLOWSLICER calls the methods that handle the lifecycle of these classes based in the following order: the *Application* class is called first, since it is the base of the whole application and is called before the other components. It is followed by the lifecycle present in the *Service* and *BroadcastReceiver* classes, as they may be called independently, with the application in the foreground or not. The next calls are to the *ContentProvider* component and finally to the *Activity* lifecycle methods.

3.3 Call Graph

FLOWSLICER uses the entry point just created to build the application's call graph in order to analyze the relationships between methods and routines and also to find out the reachable methods. Since we must not miss any possible information flow, the call graph must be built in a conservative way, i.e., at a call site we must consider every method that might be called. However, due to polymorphism and other programming languages features present in Java-based systems, the exact class of the receiver object of a virtual call site may not be known during the analysis' time. In order not to be overly conservative, FLOWSLICER uses Class Hierarchy Analysis (CHA), which exploits the fact that only subtypes of a receiver's declared type are possible types at runtime and, thus, creates edges for each of these in the final call graph. A problem that may arise from such a conservative call graph generation is related to false positives. However, we are able to discard all the found false positives after performing the dynamic analysis.

3.4 Program Slicing

According to (Weiser, 1981), a *program slice* consists only of the parts of the program that really affect the result obtained at a specific point of interesent, referred to as a *slicing criterion*. FLOWSLICER makes use of this technique in order to extract the possible flows that lead to an information leak.

The most used way of creating slices of a program is by creating a system dependency graph (SDG) (Horwitz et al., 1988). A SDG is a connected collection of every program dependency graph (PDG) (Larsen and Harrold, 1996; Malloy et al., 1994; Ottenstein and Ottenstein, 1984) related to the reachable methods of the program. The PDGs represent the dependencies between each statement of a method.

To create the edges of a PDG it is necessary to determine both the control and data dependencies within that method. A statement B has a control dependency on a preceding statement A if the output of A determines whether B will be executed. On the other hand, a statement B has a data dependency on a preceding statement A if the statement B uses the result of the statement A.

For the example of figure 1, we have that the line 4 depends on line 3 (data dependency) because line 4 uses the value of `st` defined in line 3, line 5 depends on line 4 (data dependency) because the condition's value is determined by the value of `t` computed in line 4, and line 6 depends both on line 4 (data dependency) and line 5 (control dependency) because it is using the value of `t` from line 4 and its execution is controlled by the `if`-statement in line 5. The PDG resulting from this is shown in figure 2.

The SDG is created by connecting methods the same way as they were connected in the call graph. In the example of figure 2, the nodes representing an invocation of a method should now have an edge with the respective method's PDG. The same way, the nodes that represent a *return* should have an edge to the statement that receives the result of the computation of the called method. In order to represent in-

```
protected void onCreate(Bundle savedState)
{
  String st = getSecret();
  String t = st.substring(0, 2);
  if (t.startsWith("1"))
    leak(t);
}
```

Figure 1: A simple method that may leak sensitive information.

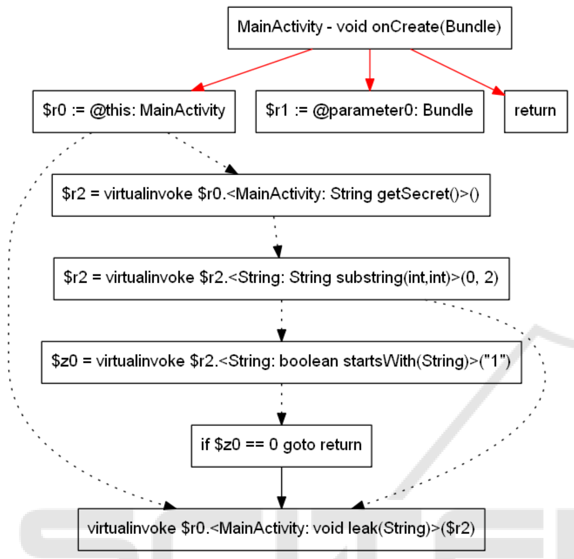
MainActivity - void onCreate(Bundle)
$r0 := @this: MainActivity
$r1 := @parameter0: Bundle
return
$r2 = virtualinvoke $r0.<MainActivity: String getSecret()>()
$r2 = virtualinvoke $r2.<String: String substring(int,int)>(0, 2)
$z0 = virtualinvoke $r2.<String: boolean startsWith(String)>("1")
if $z0 == 0 goto return
virtualinvoke $r0.<MainActivity: void leak(String)>($r2)

Figure 2: Program dependency graph for the onCreate(Bundle) method shown as example.

stance or class variables, FLOWSLICER also create new nodes to represent them. These nodes have the same dependencies as the normal statements. However, they are directly connected to the SDG and not generated inside of the PDG.

As stated before, a possible information leak takes place once the flow of an information obtained by a call to a source method hits a sink method. The same concept is used within the slice. Since the slice contains only statements that may affect the execution of a sink method, it is possible to assume that if a source method is part of the slice, we have a possible flow. FLOWSLICER uses the SDG to determine whether a statement depends on the other and, then, creates slices according to the *slicing criterion*. The use of slicing also allows the dynamic analysis to look only into the important statements, reducing caused overhead and proving the false positives not to be true.

3.5 Instrumentation

In order to help the dynamic analysis with the results obtained from the static analysis execution, FLOWSLICER makes use of *instrumentation* to point out the important statements that need to be tracked in order to determine the final sensitiveness of the objects that are used by a sink method. FLOWSLICER uses the slices obtained after using the *program slicing* technique to instrument the code in the correct points.

FLOWSLICER starts adding a statement either before or after a call to the important statements. The statements are added after a source method, since it results in an object that must be flagged as sensitive, after the statements of the flow that are in between the source and the sink calls, since they need to be analyzed regarding the sensitiveness of their result, and before the calls to the sink method, since they are going to use the possibly sensitive objects.

3.6 Tagging

Tagging is the process of defining a label, that represents a behavior or a characteristic, to objects that we want to keep track of. FLOWSLICER labels objects regarding the sensitiveness during the dynamic analysis. The instrumented calls added to the application in the previous process helps the dynamic analysis to add these labels to objects according to the need. Once an object is used in the flow, the tags are added to it and, when a sink call uses one of the sensitive objects, we are sure a leak of information has happened. In order to enhance awareness, an alert containing informations about the leak is sent to the user so that he can decide whether he should keep using the application or not.

4 EVALUATION

The evaluation of the method has been executed using a Core i7-4970 with 16GB RAM running Windows 10 and a Nexus 5X running Android 7.1.1. The decision to run the dynamic analysis on a real device instead of the emulator is motivated by problems we found while running it on the emulator, such as API calls that return `null` instead of a sensitive data. We believe this decision made the process more realistic and more trustworthy by avoiding discrepancies that may happen regarding the Android emulator execution.

The evaluation is performed with respect to two goals: First, we assess the accuracy of our approach by analysing a set of benchmark applications with known leaks. Second, we evaluate its runtime overhead, by executing both the original and the instrumented application under the exactly same conditions, i.e., using the same number and sequence of input events.

4.1 Testing Process

We start the process by running FLOWSLICER with the original application as input, obtaining an instrumented application as result. Thus, we run the original application on the device to obtain base results and as soon as it finishes, we run the instrumented version in the same conditions as previously. After that, we obtain the number of leaks found and also the performance results required to compare the overhead of our approach.

4.2 Preparing the Execution

As both original and instrumented applications will be tested equaly, we need to make sure the environment is clean to allow both to execute without any external interference. Before every execution, we uninstall any previously installed application with the same package name, preventing that any previously saved parameter or configuration interferes with the current execution. Afterwards, our test ensures that all permissions declared in `AndroidManifest.xml` are granted by the time the application is running. In both cases, the same seeds for inputing events on the device with the UI/Application Exerciser Monkey (Google, 2017) are used. We also run the `dumpsys` command repeatedly every 2 seconds to analyse memory and CPU overhead while the application is executing.

4.3 Obtaining the Results

The number of leaks found in each application is determined by running the instrumented application on a real device and using the UI/Application Exerciser Monkey (Google, 2017) to generate random events. To determine the overhead, the amount of memory and CPU consumed by the instrumented application is compared to the same values found while running the original application with the same random events as the ones generated for the instrumented one.

4.4 Testing Results

The table lists the results obtained by running the analysis for the category ANDROIDSPECIFIC from DROIDBENCH and analyze the following variables: number of real leaks (obtained from the DROIDBENCH repository), number of possible leaks found during the static analysis, number of leaks found by executing the instrumented version of the application. The last parameter considered for the result is the overhead caused by our instrumentation and dynamic analysis.

Table 1: Evaluation of FLOWSLICER for all applications contained in AndroidSpecific category of DROIDBENCH.

Application	Real	Static	Dynamic
ApplicationModeling	1	1	1
DirectLeak	1	1	1
InactiveActivity	0	0	0
LogNoLeak	0	0	0
Library	1	1	1
Obfuscation	1	1	1
Parcel	1	1	1
PrivateDataLeak1	1	2	1
PrivateDataLeak2	1	1	1
PrivateDataLeak3	2	2	2
PublicAPIField1	1	1	1
PublicAPIField2	1	1	1

4.5 Analysis of Results

As seen in table 1, FLOWSLICER is able to detect 100% of the leaks with only one false positive during the static analysis. The false positive happens because whenever a block of statements depend on some variable, all of those statements are control dependents on this variable, even though they do not use it for anything. However, it was proven not to be a leak in our dynamic analysis, that also detected all the leaks.

In order to prove that our technique not only has a high rate of detection, but also does not cause an overhead that precludes the use of the instrumented applications, we analyzed the overhead for both CPU and memory. The memory overhead caused by the instrumentation reaches an average of just 4% compared to the normal applications. It happens because we make use of memory to store the tags of the objects we need during the analysis. Given the fact that memory is typically not a scarce resource in today's mobile devices, this overhead is quite acceptable. The overhead caused on the CPU use is on average about 1%, which allows the user to use the instrumented applications without any different feeling or perception.

5 CONCLUSION

We proposed FLOWSLICER, a hybrid approach to detect, avoid and alert the user about information leak on Android devices. FLOWSLICER was able to identify all privacy leaks contained in the ANDROIDSPECIFIC category from DROIDBENCH benchmark with a small rate of false positives during the static analysis and very low overhead to the Android device while testing the application dynamically.

Despite of the results presented in this paper, there is still room for improvement. The call graph construction, although conservative, creates many different edges that end up increasing the number of false positives and raises the size of our system dependency graph drastically. The user awareness has always been the focus of this method. However, the way FLOWSLICER shows the alerts and the results of analysis is not very user-friendly. In a future work, we also want to give the user more power, allowing him to decide what is considered a sensitive information by filtering the list of sources and sinks.

ACKNOWLEDGEMENTS

The authors thank the National Council for Scientific and Technological Development (CNPq) for the financial support.

REFERENCES

Arp, D., Spreitzenbarth, M., Gascon, H., and Rieck, K. (2014). Drebin: Effective and explainable detection of android malware in your pocket.

Arzt, S., Rasthofer, S., and Bodden, E. (2013). Susi: A tool for the fully automated classification and categorization of android sources and sinks.

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

Enck, W., Cox, L. P., Jung, J., and et al. (2010). Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones.

Fan, G. and Xuan, Z. (2016). Design and implementation of a dependence-based taint analysis. In *2016 11th International Conference on Computer Science & Education (ICCSE)*, pages 985–991. IEEE.

Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. (2011a). Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA. ACM.

Felt, A. P., Greenwood, K., and Wagner, D. (2011b). The effectiveness of application permissions. In *Proceedings of the 2Nd USENIX Conference on Web Application Development*, WebApps'11, pages 7–7, Berkeley, CA, USA. USENIX Association.

Google (2017). Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey.html. [Online; accessed 09-March-2017].

Gordon, M. I., Kim, D., Perkins, J. H., Gilham, L., Nguyen, N., and Rinard, M. C. (2015). Information flow analysis of android applications in droidsafe. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society.

Hedin, D. and Sabelfeld, A. (2011). A perspective on information-flow control.

Horwitz, S., Reps, T., and Binkley, D. (1988). Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA. ACM.

Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., and Millstein, T. (2012). Dr. android and mr. hide: Fine-grained permissions in android applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 3–14, New York, NY, USA. ACM.

Kulenovic, M. and Donko, D. (2014). A survey of static code analysis methods for security vulnerabilities detection. In *MIPRO*, pages 1381–1386. IEEE.

Lantz, P. (2011). Droidbox: An android application sandbox for dynamic analysis. https://github.com/pjlantz/droidbox. [Online; accessed 27-February-2017].

Larsen, L. and Harrold, M. J. (1996). Slicing object-oriented software. In Rombach, H. D., Maibaum, T. S. E., and Zelkowitz, M. V., editors, *ICSE*, pages 495–505. IEEE Computer Society.

Livshits, B. (2005). Stanford securibench. https://suif.stanford.edu/livshits/securibench/. [Online; accessed 22-February-2017].

Malloy, B. A., Mcgregor, J. D., Krishnaswamy, A., and Medikonda, M. (1994). An extensible program representation for object-oriented software. *ACM SIGPLAN Notices*, 29:38–47.

Ottenstein, K. J. and Ottenstein, L. M. (1984). The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5):177–184.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. (1999). Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press.

Walden, J. and Doyle, M. (2012). SAVI: static-analysis vulnerability indicator. *IEEE Security & Privacy*, 10(3):32–39.

Weiser, M. (1981). Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA. IEEE Press.