Apparecium: Revealing Data Flows in Android Applications

Dennis Titze, Julian Schütte
Fraunhofer AISEC
Garching near Munich
Email: {dennis.titze,julian.schuette}@aisec.fraunhofer.de

Abstract

With Android applications processing not only personal but also business-critical data, efficient and precise data flow analysis has become a major technique to detect apps handling critical data in unwanted ways. Although data flow analysis in general is a thoroughly researched topic, the event-driven lifecycle model of Android has its own challenges and practical application requires for reliable and efficient analysis techniques. In this paper we present Apparecium, a tool to reveal data flows in Android applications. Apparecium has conceptual differences to other techniques, and can be used to find arbitrary data flows inside Android applications. Details about the used techniques and the differences to existing data flow analysis tools are presented, as well as an evaluation against the data flow analysis framework FlowDroid.

1. Introduction

The vast popularity of the Android operating system with end users and developers has also increased the interest in the platform for developers of malicious software. Antagonizing this development, much research is done into malware analysis and application analysis in general. But not only plain malicious behaviour is unwanted by users, also benign applications dealing with personal or business-critical data on the device might handle this data in an undesired way.

Apps leaking private data of a user may break legal requirements, impact the user's privacy or threaten critical business processes. At the same time, avoiding data leaks in applications is not a trivial task. Even experienced developers have currently no means of thoroughly checking all possible data flows within an application. The integration of third-party libraries into Android apps leads to large portions of code which are out of control of the actual app developer but still have access to all data of the application. There is therefore a need for reliable and efficient analysis mechanisms to detect data leaks inside applications.

Such data flow analysis techniques can be classified as either static or dynamic taint analysis.
Static data flows describe how data can flow through the app along different execution paths from a source to a sink function. Static data flow analysis detects possible data flows through the app by inspecting its code. This only states that there is a data flow which could be executed at some point in time. Further, static taint analysis is overapproximate, i.e. it considers function calls which are possible in the app, but might never occur at runtime, due to unsatisfiable conditions or lack of instantiations. That is, static taint analysis will highlight potential data flows, but if this data flows are actually executed is not necessarily known.

Contrary to that, dynamic taint analysis tracks data flows as they actually happen during execution of the app. The major and unsolved problem is that dynamic analysis relies in many cases on input from the outside (e.g., user interaction) to trigger the execution of the application. If the employed technique to provide such input does not trigger the relevant call paths, no data flow will be recorded.

So both analysis techniques have benefits and shortcomings. In this paper we describe Apparecium, a tool for static data flow analysis of Android application. Apparecium requires only the bytecode of an application but no further information such as the application entry points or its source code.

The contributions of this paper are:

- A thorough description of the basic data flow principles on Android
- An efficient static taint analysis which directly calculates flow paths between sources and sinks without the need of an entry point analysis and which is highly reliable for the analysis of real world applications.
- Providing developers of new analysis techniques

with a basic data flow analysis tool[1]

The remainder of this paper is structured as follows: Section 2 presents related work in the topic of data flow analysis. Section 3 explains how to deal with the challenge of source and sink creation, which is essential for taint analysis. In Section 4 the techniques used in Apparecium are described in detail. Section 5 briefly shows how a data flow can be visualized, and Section 6 evaluates Apparecium against FlowDroid, the prevalent static taint analysis tool, using a set of real world application. In Section 7 limitations and benefits are discussed before Section 8 concludes the paper.

2. Related Work

The most prominent example for static taint analysis on android is FlowDroid [1], an extensive data flow analysis framework built on top of Soot [8]. Soot originated as a Java optimization framework and has been extended to perform static taint analysis and to handle Android apps. FlowDroid implements an extensive taint propagation logic, covering different callbacks. FlowDroid builds upon Soot and requires an time-consuming upstream entry point analysis to gather all event-triggered callbacks in a single entry method (c.f. [1]). Differing from this approach, Apparecium follows a more efficient approach as it does not rely on the existence of a single entry point, since the analysis starts directly at the sources and sinks. Not all techniques used in FlowDroid are currently implemented in Apparecium, but there are no conceptual hurdles which would prevent the addition of these techniques. Also differing from Apparecium, FlowDroid requires the code of the app to be translated to one of Soot's internal representation, namely Jimple. This is done using Dexpler [3].

Various other scientific publications use static static taint tracking as part of their analysis:

Woodpecker [7] uses static taint analysis to detect capability leaks in stock applications. It uses its own data flow tracking implementation, which also relies on entry points into the application, which is a conceptual difference to our approach. CHEX [9] also use this approach in their data flow tracking to detect potential component hijacking vulnerabilities.

DroidChecker [4] uses static taint checking to identify data paths responsible for capability leaks, but Chan et. al. do not explain in detail how their taint tracking algorithm works.

1. Apparecium is available at github.com/titze/apparecium

AppCaulk [12] uses static static taint analysis to generate relevant points inside an application which will be instrumented with an appropriate tainting logic, so that the application itself can perform a dynamic taint analysis. AppCaulk uses Apparecium for its static taint analysis.

A conceptually different approach is taken by TaintDroid [5], which employs dynamic taint tracking. TaintDroid modifies the Android system image to track data flows as they occur during the execution of the application. To analyse an app, the modified Android OS has to be installed on a real device or the Android emulator, where the app can then be started. TaintDroid keeps track of the taint status of variables and can notify the user about leakages.

3. Sources and Sinks

An essential step for data flow analysis is the generation of sources and sinks. Since the data flow analysis can only find flows between these, a thorough configuration of these locations is essential. Best results can be expected if the sources and sinks are configured exactly to the current need: missing entries will result in false negatives, but having too many sources and sinks can slow down the analysis considerably and generate results which are not meaningful to the user.

For Apparecium, sources and sink are each configured as a list of functions. A source is a function where the data flow starts, e.g., the function `TelephonyManager.getLine1Number` which retrieves the phone number of the device. The return value of a source function is automatically marked as tainted. Similarly, a sink is a function which operates on some input, e.g., `Writer.write` which can write data to a file.

Defining sources and sinks in such a way already poses the first problem: the context of the source or sink function can not be configured. Consider the example in Listing 1. Here data is written to a Socket. If the sink should be the writing of data, `PrintWriter.print` has to be configured as sink, but this function does not state anything about the type of the PrintWriter. This can result in findings to arbitrary PrintWriters, not just such from a Socket, and therefore introduces false positives.

```
Socket socket = new Socket(IP, PORT);
OutputStream outstream = socket.
    getOutputStream();
PrintWriter out = new PrintWriter(outstream);
out.print(DATA);
```

Listing 1. Writing to a Socket

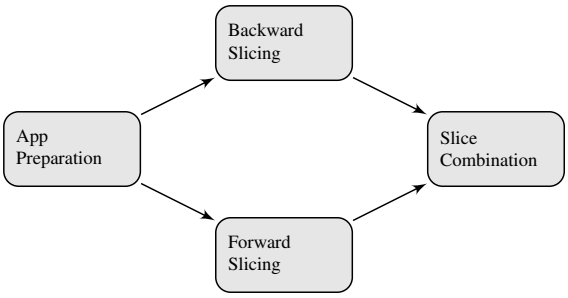

Figure 1. Sequence of Required Analysis Steps Performed by Apparecium

The second problem is even more critical: how is this list of sources and sinks generated? Since a manual configuration using a selection of all Android APIs is hardly possible, automated tools are needed. The most widely used tools for this task are SuSi [10] and the permission maps generated by Stowaway [6] and PScout [2]. SuSi was specifically designed to generate a list of sources and sinks which can directly be used for the data flow analysis.

Permission maps on the other hand were intended to provide a mapping between Android APIs and Android Permissions, which is not officially available. But this mapping can also be used as list for sources and sinks. This can be done by manually deciding which permission should be regarded as source or sink and then filling the source and sink definitions with the generated API calls. E.g., the permission READ_PHONE_STATE could be regarded as source permission, and all APIs which need this permission can then be treated as sources. But generating the list solely using the permission maps will miss several important APIs. In the example from Listing 1, according to the permission maps, the instantiation of the socket would require the permission INTERNET. But the actual call to `PrintWriter.print` is not found in the permission maps. Therefore if the permission map would solely fill the list of sources and sinks, the leak to the Internet would only be found if the ip or the port used in the Socket instantiation would contain tainted data.

As a compromise between practicability and completeness, these automated approaches can be taken as starting points, but have to be enriched by manual definitions.

4. Static Taint Analysis

This Section describes the algorithms used by Apparecium to perform static taint tracking from the defined sources to the sinks.

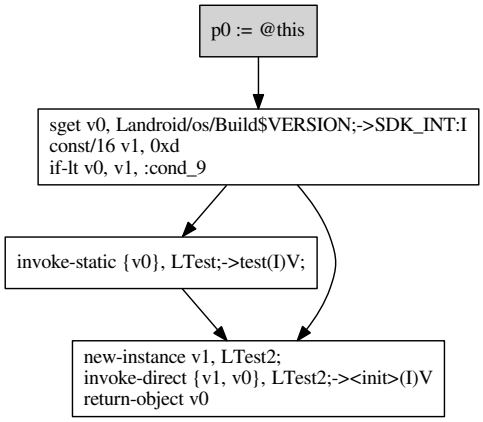

Figure 2. Basic Blocks of a Simple Function

The analysis is composed of the following steps, which are described in detail.

- App preparation
- Backward slicing
- Forward slicing
- Slice combination

4.1. App Preparation

In the first step, the app is disassembled using *baksmali*, which transforms the binary bytecode into its textual representation called *smali*. Parsing smali code allows the analysis to work directly with an app without the need for source code. Differing to other approaches, Apparecium does not translate smali to any higher language to increase efficiency and avoid semantic deviations.

As a next step, the basic blocks for each function are generated. A basic block is a sequence of instructions which is executed without any branches. For example, as soon as there is a branching statement in the function, it marks the end of the basic block and the two successors of that statement (the true and the false branch) are the first statements of two further basic blocks. Figure 2 shows a simple function split into its basic blocks.

In the app preparation step, a class hierarchy is also constructed from the smali code, which is subsequently needed to determine the callers of functions.

4.2. Backward Slicing

After the preparation step, *backward slicing* analyses the prepared bytecode and generates for each line of code a list of variables which can reach a sink. That is, if variable `v0` is in this list, there exists a data flow were the data from `v0` reaches a sink.

The analysis starts at all sinks and adds these locations to a worklist. The algorithm runs as long as there are entries in the worklist. An entry consists of a pointer to the function, a pointer to the location inside the function (i.e., the program counter), and a call stack.

During the backwards analysis, the algorithm may encounter a function call and therefore needs to jump to that function to determine if the taint status is propagated or not. In such a case, the call stack records the current function and program counter. This is the location where the analysis will continue once the called function is analysed. Since this function itself can contain new function calls, the call stack can record multiple function locations.

If the function to be examined exists in the current app (i.e., it is part of the application's code base), the analysis simply continues at this function. If the function is out of the scope of the application's code base (e.g., for framework functions or native functions), Apparecium overapproximates the taint propagation and assumes that the function propagates the taint status.

In each iteration, the backward slicing algorithm takes an element from the worklist and determines if the current instruction propagates the taint status. If at least one of the parameters of the instruction is currently tainted, it has to be decided if any input parameter will also be tainted. Consider the instruction `move v1, v2` which moves the content of variable `v2` into variable `v1`. If variable `v2` is tainted in this location, the input variable `v1` will be tainted before this location. How the tainting is propagated can be easily configured and has to be done accurately for each Dalvik instruction. An instruction can also untaint a variable, e.g., overwriting it with a constant value (i.e., `const v0, 0`). If the backward analysis encounters such an instruction, it removes the variable from the currently tainted parameters.

If any input variable is tainted (there could be multiple new variables tainted), the previous location needs to be determined. There exist three possibilities for previous locations:

- the previous instruction,
- the previous basic block, or
- the caller of the current function.

If the current location is not the beginning of a basic block, the previous location is simply the previous line of code. If the analysis reaches the beginning of a basic block, the predecessors of the basic block need to be determined. If the current basic block is not the first basic block of the function, the analysis continues at the last instruction of predecessors of the current basic block. If the current basic block is the first basic block of the function, no precessing block exists. The analysis will therefore either continue at the top element of the call stack, if available, or at all locations inside the application where the current function is called. Therefore all locations which can call the current function are added to the worklist. Currently direct function calls and function calls using any assignable class are considered. Considering all assignable classes is an overapproximation, but does not add any false negatives.

In the last step, the algorithm checks if the tainted variables in the previous locations already contain the newly tainted variables. If no, these variables are added to the locations and the previous locations are added to the worklist. If the previous location already contains all currently possible tainted variables, it has been already added to the worklist at some point. In such a case, the locations are not added to the worklist again.

The backward slicing will terminate once no element remains in the worklist. Since there exist only a finite amount of lines of code with a finite amount of variables, the algorithm will terminate.

4.3. Forward Slicing

The *forward slicing* performs a similar analysis than the backward slicing, by generating a list of variables which can contain data from a source. The algorithm works in the opposite direction of the backward slicing: starting at all sources, an analysis is conducted to find all paths originating in one of these sources which propagate the taint status. To do so, the sources are added to a worklist. The worklist contains entries consisting of the current function, the current program counter and the call stack. The call stack serves the same purpose as in the backward analysis: it is used to store the caller of a function to know where to return to at the end of a function.

In each iteration of the forward slicing algorithm, an element is taken from the worklist and it is determined if the taint status is propagated from the input parameter to the output parameter of the instruction. Considering the easy example `move v1, v2` again, the taint status from `v1` is propagated to the variable `v2`. As in the backward slicing, if the instruction

untaints the variable, it is removed from the currently tainted variables.

If at least one variable is in the currently tainted variables in the current line, the next instruction has to be determined. The next location can be either:

- the beginning of a function if the current instruction is a function call,
- the next instruction inside the current basic block,
- the next basic blocks, or
- the return address of the function.

If the current instruction is a function call, the algorithm has to determine if the function propagates the taint status of an input variable of the function to an output variable of the function (e.g., for the function a = sqrt(b), b would be an input variable, a would be an output variable). Therefore the algorithm adds the beginning of the function to the entries in the worklist, with the input variables tainted.

If the next location is an instruction inside the current basic block, the algorithm simply copies the list of currently marked variables to the next instruction, adds the instruction to the worklist and continues.

If the current location is the end of a basic block, the next basic blocks have to be determined. This can e.g., be the two basic blocks following an if statement, or an additional catch block for exception handling. The algorithm again copies the currently tainted variables to the start of these basic blocks and continues.

If the current location is the return statement of the function, and the call stack is not empty, the algorithm propagates the status of the returned variable to the caller. The returned taint status can either be the status of the actual returned variable, or the current instance of the class, if the class itself is tainted (i.e., the this-variable inside the function was tainted). If no entry exists in the call stack, the algorithm has to take all possible callers of the function into consideration. Analogous to the backward slicing, possible callers are direct function calls and function calls using any assignable class.

The forward slicing algorithm terminates once the worklist is empty. As with the backward slicing, the forward slicing will terminate since there exist only a finite amount of lines of code with a finite amount of variables.

4.4. Extending the Call Graph

To perform accurate data flow analysis the call graph has to be as exact as possible, since both backward and the forward slicing follow the paths in the call graph extensively. A missing path in the call graph can therefore lead to undetected data flows. For this reason, several paths need to be examined which are not part of the call graph. The call graph is therefore extended by several new edges and becomes a Data Flow Graph (DFG).

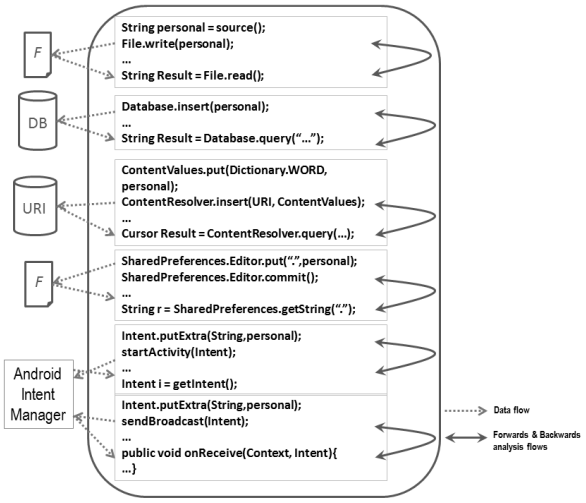

Figure 3. External data flows

```
class A {
  public static int STATICVAR;
  public void a() {
    STATICVAR = source();
  }
  public void b() {
    sink(STATICVAR);
  }
}
```

Listing 2. Data Leak through a Static Variable

Consider the example in Listing 2. If Apparecium would only look at the default call graph, the backward analysis would start in function `b()`, but since there is no caller of `b()`, the analysis would simply stop. But an app can contain this snippet and have some user interaction capabilities, where at some point in time function `a()` is triggered, and at some other time function `b()`. This would result in an undetected data leak. To solve this problem, an edge from all writing locations of static (or instance) variables to all reading locations is added to the DFG. If the backward analysis therefore encounters the reading of such a variable, it can determine all writing locations and continue its analysis there.

New paths can be easily added to the DFG in Apparecium. To do so, the reading and writing locations have to be linked, e.g., the reading with the writing of files. Callbacks which are used extensively in Android can be added in the same fashion.

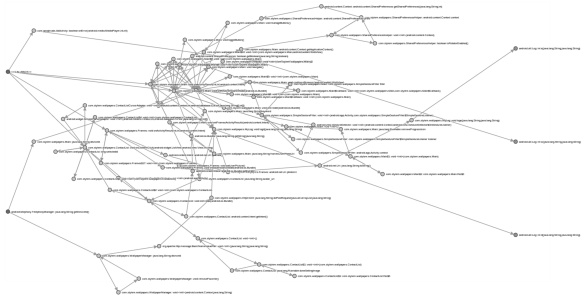

Figure 4. Data Flow Visualization on Function Level

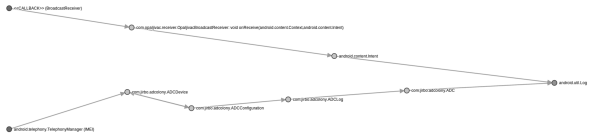

Figure 5. Data Flow Visualization on Class Level

Figure 4.4 shows the external call paths currently included in Apparecium.

Adding flows for file read and writes to the DFG may discover more actual data leaks, but it will also increase the false positives rate. For instance, if data is written to a file and read at another location in the app, an actual data flow occurs only if at both location the same file content is accessed, i.e. if it is the same file and it has not been modified in the meanwhile. Tracking these conditions is only possible at runtime, since access to files may happen outside of the analysed application scope. One optimization would be to heuristically determine the instance of the file handler, which could reduce the number of false positives.

4.5. Slice Combination

After generating both the backward and forward slice, these can be combined to show the complete data flow from sources to sinks. The combination simply iterates over all locations and adds a variable to the final result if it is contained in both the forward slice and the backward slice.

Since the backward slice contains list of variables for each location which can reach the sink, and the forward slice contains the list of variables which can contain data from a source, this final result contains all locations with variables which can be part of the data flow.

FlowDroid	
Average Runtime	815 Seconds
Total Runtime	1359 Minutes
Findings	10^3
Out of Memory Errors	50
Other Errors/Exceptions	30
Timeout after 1 hour	5
Apparecium	
Average Runtime	1175 Seconds
Total Runtime	1959 Minutes
Findings	68
Out of Memory Errors	0
Errors	1
Timeout after 1 hour	17

Table 1. Statistics of the Evaluation

5. Visualization

The data flows generated by Apparecium can be visualized for further examination by the user. This is done by transforming the textual output of Apparecium into a D3[2] graph.

The visualization displays the data flow on different levels of detail. Figure 4 shows a data flow on the highest level of detail: each node represents one function through which the data flows from a source on the left side of the figure to a sink on the right side. If there are many data flows the visualization on function level will contain too many nodes to provide any benefit. For this reason, the level of detail can be reduced, so that each node only represents a class participating in the data flow. The sources and sinks are still displayed as functions, to be able to see the actual function. Figure 5 displays a data flow on this lower detail level.

6. Evaluation

While different tools for static taint analysis of Android apps exist, our goal was the implementation of an approach which is highly practical and applicable to real world applications. We therefore collected 100 random apps among the most popular ones from the Google Play Store and evaluated the effectiveness and efficiency of Apparecium, compared to FlowDroid, which is currently the prevalent and most mature static

2. http://d3js.org/

3. FlowDroid was able to finish some apps even though there were Exceptions. The 10 findings therefore were not necessarily in apps which did not contain any errors.

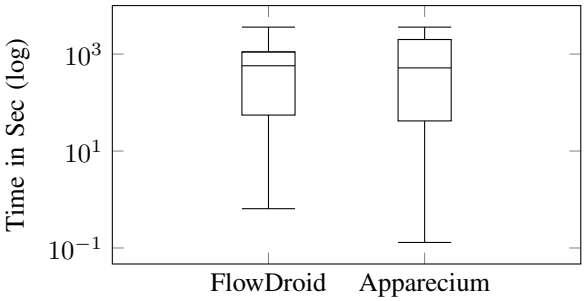

Figure 6. Runtimes of Apparecium and FlowDroid[5]

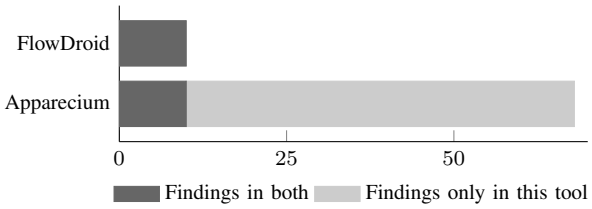

Figure 7. Data Flows found in Apparecium and FlowDroid

taint analysis tool for Android. The app selection includes a range of different apps, ranging from a very small flashlight app, to complex apps like Facebook and WhatsApp. FlowDroid is used in the configuration from [1], and both tools are executed sequentially on the same PC (Intel Core i7-3520M) with 4GB of RAM assigned to them.

To reflect the usage of both tools in a productive environment such as an in-app security check or a testing tool, a timeout of one hour was set for both. The lists for sources and sinks were filled with the same inputs, namely the default sources and sinks from FlowDroid.

Basic details of the evaluation are shown in Table 5, details about the run times are shown in the box plot in Figure 6 and the details about the determined data flows in Figure 7.

In total, Apparecium needed 1959 minutes to complete the analysis compared to 1359 minutes of FlowDroid. But the total runtime cannot be used as a criteria of quality, since more than half of the applications could not be analyzed by FlowDroid due to the memory constraint.

Comparing the results of the findings (c.f. Figure 7) shows that 10 apps with data leaks were detected by both Apparecium and FlowDroid. A comparison of the apps which contained a data flow showed, that all 10 apps which were found by FlowDroid were also found by Apparecium. For 58 of the apps, a data flow was only found with Apparecium. This is on the one hand due to the fact, that many applications could not be analysed completely using only 4 GB of RAM and the one hour time limit. On the other hand FlowDroid already performs optimizations for truncating false positives, which are currently not included in Apparecium.

Table 5 shows that the high accuracy of FlowDroid comes at a cost: half of the 100 apps could not be analysed with only 4 GB of RAM by FlowDroid. This suggests that FlowDroid soon exceeds practical resource limits when applied to applications of real world size and complexity. In contrast Apparecium is able to cope with a significantly larger number of applications and identifies many more data flows. It must however be considered that the higher number of findings is not only due to the higher reliability of Apparecium but also due to its higher false positive rate.

5. The comparison of runtimes has only limited significance, since FlowDroid was not able to analyse more than half of the apps due to memory constraints.

7. Discussion

In its current implementation, Apparecium is missing some optimizations, e.g., the addition of more callbacks to the DFG (as described in Section 4.4), or the identification of the instances responsible for the external path which would reduce the number of false positives. But these limitations do not impose a conceptual limitation in the proposed static taint analysis.

A different problem – which is currently not solved in other tools as well – is caused by dynamic code loading. Since such loading can occur at arbitrary locations in the code, dynamically loaded code is not available in an offline static analysis. To tackle this problem, different approaches have been proposed, e.g., the concolic execution of the app [11]. The same is true for native methods, which are not included in the analysis.

The third limitation arises from the fact that Apparecium overapproximates at several points, e.g., by adding all function calls of assignable classes, and including external call paths. Although this increases the potential false positives, it also allows us to find more data flows which might be missed otherwise.

Overapproximation also implies that finding a data flow inside an app does not necessarily mean that this data flow will be actually executed. Since Apparecium does not perform its taint analysis using entry points into the app, identified data flows can for example take

place in dead code or require instantiation of objects which do not occur at runtime. Nevertheless, finding such flows is relevant, since the app can use different techniques to execute such code, e.g., via reflection.

8. Conclusion

In this paper we presented Apparecium, a tool to statically detect data flows in Android applications from arbitrary data sources to sinks. The aim of Apparecium is to provide efficient and highly practical static taint analysis to discover data leaks in Android applications. Unlike other approaches, Apparecium does not rely on an expensive entry point analysis, and performs its static taint analysis directly from sources to sinks, thereby achieving a high level of efficiency. By means of an evaluation against 100 of the most popular applications from the Google Play Store we have shown that Apparecium is in fact able to successfully analyse more applications than current static data flow tools.

In addition, we discussed general challenges in static taint analysis for Android, such as a highly precise context-based definition of sources and sinks which is currently not possible and motivates further work on the topic.

References

[1] 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 *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[2] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

[3] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, SOAP '12, pages 27–38, New York, NY, USA, 2012. ACM.

[4] P. P. Chan, L. C. Hui, and S. M. Yiu. Droidchecker: Analyzing android applications for capability leak. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, pages 125–136, New York, NY, USA, 2012. ACM.

[5] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *9th USENIX conference on Operating systems design and implementation*, 2010.

[6] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[7] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, 2012.

[8] P. Lam, E. Bodden, O. Lhotak, and L. Hendren. The soot framework for java program analysis: a retrospective. In *CETUS Users and Compiler Infrastructure Workshop*, Oct. 2011.

[9] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[10] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *2014 Network and Distributed System Security Symposium (NDSS)*, 2014.

[11] J. Schütte, R. Fedler, and D. Titze. Condroid: Targeted dynamic analysis of android applications. In *(in review)*, 2014.

[12] J. Schütte, D. Titze, and J. M. d. Fuentes. Appcaulk: Data leak prevention by injecting targeted taint tracking into android apps. In *Proceedings of the International Conference on Trust, Security and Privacy in Computing and Communications*, TrustCom, 2014.