Static Taint Analysis Tools to Detect Information Flows

Dan Boxler
University of Colorado, Corlorado Springs
Colorado Springs, CO 80918
Email: dboxler@uccs.edu

Kristen R. Walcott
University of Colorado, Corlorado Springs
Colorado Springs, CO 80918
Email: kwalcott@uccs.edu

***Abstract*—In today's society, the smartphone has become a pervasive source of private and confidential information and has become a quintessential target for malicious intent. Smartphone users are bombarded with a multitude of third party applications that either have little to no regard to private information or aim to gather that information maliciously. To combat this, there have been numerous tools developed to analyze mobile applications to detect information leaks that could lead to privacy concerns. One popular category of these tools are static taint analysis tools that track the flow of data throughout an application.**

In this work we perform an objective comparison of several Android taint analysis tools that detect information flows in Android applications. We test the tools FlowDroid, IccTA, and DroidSafe on both the android taint analysis benchmarking suite DROIDBENCH as well as a random subset of applications acquired from F-Droid and measure these tools in terms of effectiveness and efficiency.

Keywords: Taint Analysis, Information Flows, Static Analysis, Mobile Development

I. Introduction

As mobile devices become increasingly popular, there has been a shift in development for these platforms. Devices such as smart phones and tablets have taken over the forefront of mobile development over laptops and netbooks. There are several mobile operating systems out there but the most prominent in terms of market share is by far the Android operating system developed by Google [3]. With its increasing popularity, Android has become a more attractive target for malicious developers that wish to obtain private information from users. This could be anything from location data, contact information, passwords, or even user activity.

Software applications on Android can be acquired from virtually anywhere and there are several marketplaces where users can find and download applications. Unlike other mobile operating systems such as iOS, there is no approval process that an application needs to go through to be uploaded to a marketplace such as Google Play. Thus, there is no barrier to entry for a developer to create an application that has malicious intent. Android attempts to address this issue by requiring application developers to declare what restricted resources they intend on using in their applications in a manifest that the user sees and must agree to prior to downloading the application. This helps in informing the user on what restricted resources the application has access to, but it does little to inform them how the resources are being used.

Furthermore, there is a theme of many applications to request more permissions than is actually needed for proper execution of the application [10]. A large number developers ask for more permissions and violate the least permission principle in favor of not needing to request additional privileges on future updates. Most marketplaces such as Google Play rely on user regulation to filter out unsavory applications based on reviews and reports of malicious behavior.

A common trend in Android malware are privacy leaks. A privacy leak occurs when a malicious application sends private information from a device to an external destination without the user's consent or knowledge. This can be anything from contact information, location details, passwords, or any data that could compromise the user's privacy. This information can be used by malicious users ranging from anything from targeted advertising to identity theft.

There are many static analysis tools that have been developed to detect for information flows that may lead to privacy leaks in Android applications. Static analysis is a testing technique that involves inspecting code without execution. Tools will analyze all potential paths in the program looking for specific vulnerabilities or defects. Static taint analysis involves tainting a piece of information from where it originates so that it can be found during subsequent analysis. Android static taint analysis tools analyze data flow from an information source where private information is generated or stored to an information sink where the data leaves the device to a potentially malicious destination. These results can then be manually analyzed to determine if the flow is indeed malicious or not. Alternatively, there are also malware detection tools that can assess the results for malicious intent.

While there are many tools out there that intend to detect information flows in Android applications, there exists no objective comparison for such tools. It is necessary for a comparison to exist if there is to be a marketable application for these tools in industry. For example, if there was a screening process for applications to be added to a marketplace in that they need to not leak any private info, a static analysis tools could be used to determine if an application is safe for users. A tool of that nature would need to both fulfill the requirements of determining privacy leaks as well as do so in a cost efficient manner. To address this issue, we present an empirical analysis of three of the most state of the art tools for

static analysis of information flow detection. In this analysis we will focus on effectiveness and efficiency. Effectiveness will be the tools ability to detect data flows in a Android application and efficiency will be measured by the total run time for a tool to run its analysis on an application and the amount of resources it consumes in the process. This will be useful in determining the cost of running each tool. Our contributions are as follows:

- Design a methodology for testing static analysis tools for mobile applications
- Develop a third-party test suite from various Android marketplaces and compare results of the chosen static analysis tools.
- Objectively compare various Android static analysis tools using pre-established benchmarking suite DROIDBENCH in terms of efficiency and effectiveness.

II. Challenges in Malicious Behavior Detection

As users are becoming more dependent on mobile devices throughout their everyday lives, they are being heavily relied upon to store personal data. This data has many privacy concerns when it is used in mobile applications with or without the user's knowledge. The user essentially has to trust that the application is using his/her data in a secure and appropriate way. The caveat is that many applications appropriately use this information and their behavior is difficult to distinguish compared to that of a malicious application.

For example, if there is an application that provides the user a list of nearby businesses that correspond with a search query, it would need to obtain the location of the device using the GPS sensor and provide that to a server to provide results. This would be a justified use of a restricted resource and should not be flagged as malicious behavior. Alternatively, if the application also provided the location data to another source such as an advertisement database or another 3rd party for targeted advertising, this should be flagged as malicious.

As an extension, an alarming percentage of users do not exhibit the comprehension or attentiveness to make sound decisions on whether a particular application needs the permissions it has requested [11]. If a user has downloaded an application from an Android marketplace, there is a strong likelihood thatthe user will not be able to determine whether or not to cancel the installation based on the permissions and possibility of malicious execution.

There are several tools that attempt to detect such potential malicious behaviors using static analysis. However, there currently exists no comparison between them, which leaves a user at a loss when it comes to choosing the correct tool to use. Furthermore, there exists no precedent of how a new tool in this realm should be evaluated. While there exists a benchmark of hand-tailored applications that exhibit the most common information flows, originally proposed by the creators of FlowDroid [6], there is not formal comparison that puts the benchmark to use. The aim of this paper is to analyze various tools and determine the effectiveness and effectiveness of finding information flows in Android applications.

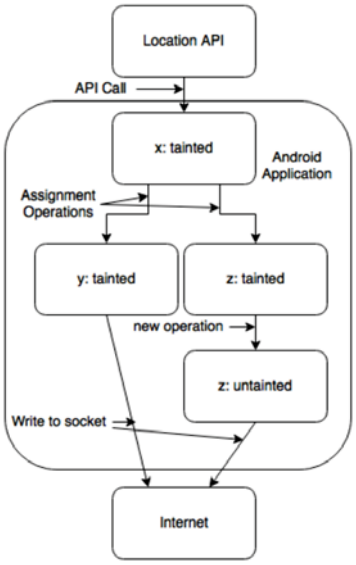

Fig. 1: Taint Analysis Example

III. Information and Taint Analysis

This section discusses the various aspects involved in the detection of information flows and potential privacy leaks as well as the tools that are used for our analysis and comparison.

A. Information Flow

An information flow tracks the data stored inside of an object x and then transferred to another object y. As we are detecting information flows in Android applications, we define information flow as the transfer of information from an information source such as accessing a GPS location API to an information sink such as the application writing to a socket. In general, this refers to information being obtained by an application and then sent outside of the application, either to another application or the Internet. Sensitive data acquired by an application through an API can be potentially leaked maliciously through an information flow to an information sink. Without a valid information flow, sensitive information cannot leave the application or a pose a risk for users.

B. Taint Analysis

Taint analysis is used to track information flows from an information source to a sink. Data derived from an information source is considered tainted while all other data is considered untainted. An analysis tool tracks this data and how it propagates throughout the program because tainted data can also influence other information through assignments or other operations. For example, a left-hand operand of an assignment statement will be tainted if and only if one or more of the right-hand operands were previously tainted. Taint can be revoked under circumstances such as a new expression or assignment.

Figure 1 shows an example of how taint analysis works. A variable x is tainted based on the fact that it is assigned

information from a location API call. This taint is then propagated to variables *y* and *z* due to assignment operations with *x* as a right-hand operand. Variable *z* is then untainted by performing a new operation which effectively erases the tainted information from the variable. Lastly, the data in variables *y* and *z* are written to a socket to a location on the Internet. Only the writing of the data within y represents an information flow since it still contains tainted information. As such, a taint analysis tool will report this flow to the user as it potentially represents a privacy leak in the application.

There are two contrasting techniques that can be used for taint analysis: static taint analysis and dynamic taint analysis. Three tools are known to support these analyses: FlowDroid, IccTA, and DroidSafe.

Static Taint Analysis is done outside the operating environment and is run using a tool on a computer. Static analysis tools are able to provide better code coverage compared to dynamic analysis tools but suffer in that they are unable to access the runtime information of the intended environment since it is running on a different platform.

Dynamic Taint Analysis is run while a program is executing. It is often run on the same hardware and while the programming is actually being run in the intended environment. Since it is running during program execution, it has access to all of the runtime information and can determine flows that are inherently too complex for static analysis tools. The downside of dynamic analysis tools is that they are only able to find flows that are actually being executed. Therefore, they often have less code coverage than static analysis tools.

FlowDroid: FlowDroid [6] is an extension of the SOOT framework [22] that is able to precisely detect information flows in Android applications. The Soot framework is a compiler extension for Java applications. It takes as input either Java source code or Java bytecode and outputs Java bytecode. It was originally developed as an optimizer for Java programs. The main features of SOOT are its intermediate representations of the Java code that are produced, in particular Jimple, which is a typed three-address code. In addition, Soot also provides callgraph and pointer information, which are essential to developing a static anal ysis architecture.

FlowDroid extracts the contents of an Android .apk file and parses the Android manifest, XML, and .dex files that contain the executable code. Using the tool SuSi [5], the sources and sinks of the application are identified and the callback and lifecycle methods are determined. From this, a dummy main method is generated. This is used to generate a inter-procedural control flow graph which starts at sources where data is tainted and then followed to sinks to determine flows in the application.

IccTA: Inter-Component Communication Taint Analysis (IccTA) [18] focuses specifically on detecting privacy leaks between components within Android. IccTA uses Dexpler [7] to convert Dalvik executables into Jimple, which is the representation used internally by the Soot [16] framework, which is popular for analyzing Android applications. They then utilize Epicc [18] and IC3 [20] to extract the inter-component communication (ICC) links from an application and combine that with the Android intents that are associated with them. They then modify the Jimple representation to connect the components that are involved in the ICC so that they can be analyzed with data-flow analysis. With this enhanced model of the application, the process employs an altered version of FlowDroid to detect flows.

DroidSafe: DroidSafe [13] is another system for detecting information leaks in Android applications. DroidSafe utilizes the Android Open Source Project [1] which is a precise model of the Android environment and use a stubbing technique to add in additional parts of the Android runtime that are imported for their analysis. This model is then used with static analysis techniques that are highly scalable to detect information leaks in the target Android application.

IV. Experiment Setup

This section describes the experiment setup and and design for our evaluation of the Android static analysis tools.

A. Experimental Design and Metrics

The experiments were executed on a GNU/Linux workstation with kernel version 4.2.0-16 with an Intel(R) Core(TM) i5-4670K CPU @ 3.40GHz with 16 Gb of RAM.

1) Test Suites: There are two test suites used to benchmark the various Android static taint analysis tools, DROIDBENCH, and F-Droid: **DROIDBENCH** is a suite Android applications that were originally developed by the creators of FlowDroid with the intent of creating a Java-based benchmark suite with the same aim as SecuriBench [19] which was designed for Java-based web applications. In our experiments, we utilize DROIDBENCH version 2.0 which contains 119 Android applications that were specifically written with the intent of testing Android taint analysis tools. while this suite of applications can be used with both static and dynamic taint analysis tools, many of the applications address the shortcomings of static analysis tools such as field sensitivity and object sensitivity. Also, there are also applications that focus on Android specific issues such as asynchronous callbacks and user interface interactions. DROIDBENCH has been used in the development of many Android Taint analysis tools including [6], [8], [13], [17], [21].

We consider thirteen classes of applications within the DROIDBENCH suite: Aliasing, Android specific, Arrays and lists, Callbacks, Emulator detection, Field and object sensitivity, General Java, Implicit flows, Inter-application communication, Inter-component communication, Life cycle, Reflection, Threading.

In addition to being specifically developed to test taint analysis tools, each application includes annotations that denote how many information flows exists in the application. This information is used in the evaluation of the tools in our experiments to determine a tool's effectiveness.

F-Droid [2] is a repository of Free and Open Source Software (FOSS) Android Applications. F-Droid was founded in 2010 by Ciaran Gultnieks and now hosts over 1,800

applications available for download on Android devices. All of these applications have the source code freely available to the public. Of these applications, 50 randomly chosen applications were used in our experiments in comparing the static taint analysis tools. These applications are used to compare the tools on larger Android applications.

2) Evaluation Metrics: Each static taint analysis tool is evaluated in terms of efficiency and effectiveness. Efficiency is measured based on execution time for each tool. The other measure is that of effectiveness which will differ based on which application suite we are considering.

In the DROIDBENCH application suite, there are a predetermined number of information flows that a given application will exhibit. Knowing this, we can perform several computations to calculate the effectiveness of a tool. In order to calculate these metrics, it is first necessary to obtain the true positives T_p , true negatives T_n , false positives F_p, and false negatives F_n as identified by each of the tools. The first measure is precision, which is the ratio of correctly identified information flows over the total number of identified flows and can be modeled by the following:

$$precision = \frac{T_p}{T_p + F_p}$$

The next measure is recall, which can be defined as out of all of the actual information flows in the applications suite versus how many of those flows were identified. Recall can be modeled by the following:

$$recall = \frac{T_n}{T_p + F_p}$$

The third measure is accuracy, which is defined as the ratio of true results to the total number of cases analyzed and can be modeled by the following:

$$accuracy = \frac{T_p + T_n}{T_p + F_p + T_n + F_n}$$

The last measure is F-measure, which combines both precision and recall together. It provides a weighted average between precision and recall. In this work we employ the balanced F-score metric (F1) which is the harmonic mean of precision and recall which is modeled by:

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

In terms of the F-Droid application suite, since we do not know the number of information flows that actually exist in the applications, we are limited to only comparing the static analysis tools based on the raw number of flows reported.

B. Experiment Implementation

Three Android Static Taint analysis tools were chosen for this work: FlowDroid, Inter-Component Communication Taint Analysis (IccTA), and DroidSafe. The three tools are written in Java and capable of running on an Android .apk file and focus on solving the same problem of detecting information flows in Android applications. They also all require the Android platform to be imported in order to analyze the applications. For consistency, the Java Virtual Machine (JVM) used to run each tool is allocated 16 gigabytes of RAM.

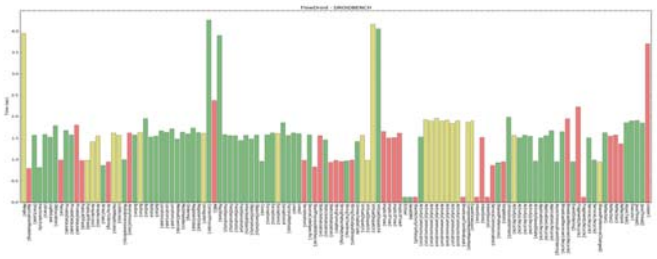

Fig. 2: FlowDroid - DROIDBENCH

In order to perform the comparison, a tool was written in Python 3. This tool sets up the necessary environments for each tool and runs them on an input set of Android applications. The tools runs the tool and times the execution time for each run. The tool then parses the output for a run and extracts the number of information flows that were reported by the tool. In regards to the DROIDBENCH application suite, these reported results are subsequently compared with the actual number of information flows that exist in the application as reported by the authors. This is used to determine whether or not the tool successfully discovered all flows and whether or not there were false positives false negatives reported. All executions were terminated if they took longer that 2 hours as some applications caused several of the tools in question to reach a state in which they would never finish.

The timing results will be useful for comparison of the tools in terms of cost to an organization that wishes to utilize these tools. They may not be helpful to a developer that wishes to analyze their personal application, but to a company such as Google that wishes to utilize this tool on a large scale, the time it takes to run a taint analysis tool for screening uploaded Android applications correlates strongly with a cost of operation. The longer an application takes to run can impose a large cost when applied to a large scale.

The effectiveness measure, which is how many applications in which a tool successfully detects an information flow is also very important in terms of comparison. The more effective a tool is demonstrates its worth in both research and industry.

These results are then stored JavaScript Object Notation (JSON) format for use in analysis and graph generation for a clear comparison between taint analysis tools. Graphs are generated using the Python modules NumPy and Matplotlib .

V. Evaluation

Figures 2, 3, and 4 display the results of the tools being executed against the DROIDBENCH application suite. Each bar of the graphs represent the execution time of a tool against a particular application of the test suite. It should be noted that due to large difference in execution times between the various tools, the scaling is not the same between each of

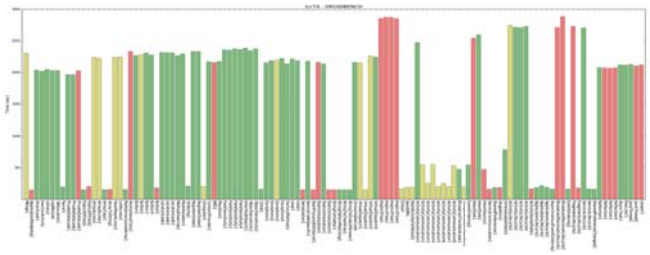

Fig. 3: IccTA - DROIDBENCH

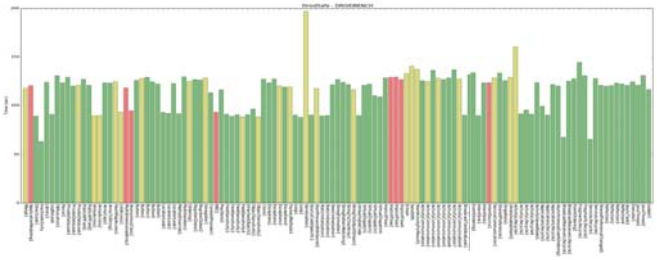

Fig. 4: DroidSafe - DROIDBENCH

the graphs. The color of the bars also indicate a measure of success in the execution. A green bar indicates that the tool was able to successfully identify all information flows from the application. A yellow bar indicates that the tools identified more flows in the application than were intended, leading to one or more false positives in the results. A red bar indicates that the tool was unable to identify one or more information flows that existed in the test application.

Figure 2 illustrates the results of FlowDroid on the DROIDBENCH test suite. It reveals that the execution time of running FlowDroid over the application suite ranged from 0.12 seconds to 4.27 seconds. It also highlights that many of the applications that applications such as those in the intercomponent communication category were over estimated in the number of flows leading to false positives. Also, it is evident that FlowDroid was unable to successfully discover flows in the implicit flow and reflection categories of applications.

Figure 3 highlights the results of IccTA on the DROIDBENCH test suite. The execution time of running IccTA on the DROIDBENCH suite ranged from 15.26 seconds to 288.80 seconds. This is substantially higher than that of FlowDroid and also exhibits the largest range of execution times out of any of the other static taint analysis tools. By observing the chart, we can see that there are many similarities in the results between IccTA and FlowDroid. This can be somewhat expected since IccTA uses a modified version of FlowDroid in their implementation. However, it is surprising that IccTA had many false positives in the intercomponent communication applications such as Activity- Communication2.apk since IccTA aims at addressing flows that cross multiple components.

Figure 4 shows the results running DroidSafe on the DROIDBENCH test suite. The execution time for DroidSafe on the application suite ranged from 63.54 seconds to 196.94 seconds. The execution times for DroidSafe were much more

	FlowDroid	IccTA	DroidSafe
True Positives	57	61	81
True Negatives	13	11	10
False Positives	23	28	38
False Negatives	38	35	12
Flows Reported	98	106	139
Precision	71.2%	68.5%	68.1%
Recall	60%	63.5%	87.1%
Accuracy	53.4%	53.3%	64.5%
F-Score	0.65	0.66	0.76

Fig. 5: Statistics for DROIDBENCH

predictable that that of FlowDroid and IccTA. DroidSafe also had much better results in terms of the inter-component communication applications and was able to correctly identify most of the intended flows in those applications. DroidSafe, along with the other tools was unable to identify the information flows in the implicit flow category. This may need to be a focus for future research in the area.

Figure 5 breaks down the results for the three Android static taint analysis tools when run against the DROIDBENCH application suite. True positives represent information flows that are explicitly declared in the various applications that were successfully reported by the tools. True negatives represent the number of applications that contained no information flows and were reported as such. False positives represent flows that were reported by a tool that did not actually exist in the application. False negatives represent when a static analysis tool reports that there were no flows present in the application when there were flows declared. Using these statistics, we were able to derive the precision, recall, accuracy, and F-score for each tool based on the DROIDBENCH benchmark suite.

Precision, which is the fraction of reported flows that are indeed existing flows with respect to the number of incorrect flows reported, for the three analysis tools is 71.2%, 68.5%, and 68.1% for FlowDroid, IccTA, and DroidSafe respectively. Despite reporting the least amount of actual information flows, FlowDroid was in fact the most precise tool, albeit by a small margin. IccTA and DroidSave also had very similar precision results. In terms of recall, which is the ratio of true reported flows against the number of reported flows, 60% for FlowDroid, 63.5% for IccTA, and 87.1% for DroidSafe. DroidSafe had by far the highest recall out of any of the tools which is directly correlated with the fact that it reported a significantly larger number of information flows.

In terms of accuracy, both FlowDroid and IccTa had similar results with 53.4% and 53.3% respectively. DroidSafe, on the other hand, had a much higher accuracy of 64.5%. This theme also holds true in terms of the F-score, which is a weighted average between precision and recall where Droid- Safe had a much higher value of 0.76 compared to FlowDroid and IccTA (0.65 and 0.66).

Figure 6 displays the execution statistics for each of the taint analysis tools when run against the DROIDBENCH applications. FlowDroid was by far fastest tool with an average

Tool	Average	Median	Minimum	Maximum
FlowDroid	1.55 sec	1.58 sec	0.12 sec	4.27 sec
IccTA	151.03 sec	211.60 sec	15.26 sec	288.80 sec
DroidSafe	151.80 sec	122.01 sec	63.53 sec	196.94 sec

Fig. 6: Execution time for DROIDBENCH

Tool	Average Time	Flows Reported	Failures
FlowDroid	99.83 sec	39	1
IccTA	384.16 sec	137	4
DroidSafe	393.80 sec	384	40

Fig. 7: Results for FDROID

execution time of only 1.55 seconds and a maximum of 4.27 seconds. IccTA and DroidSafe had very similar average execution times with 151.03 seconds and 151.80 seconds respectively but the amount of time that it took to complete for DroidSafe was much more consistent. IccTA was very inconsistent in terms of execution times with some applications only taking 15-16 seconds while others taking close to 5 minutes while DroidSafe never took longer than 3.5 minutes.

Figure 7 presents the results from running all three tools on the F-Droid application suite. The average execution time only takes into consideration the runs of each tool that did not result in a timeout or a failure. This experiment shows that FlowDroid once again executed significantly faster than the other two tools. IccTA and DroidSafe took a similar amount of time to complete although IccTA had a considerably lower number of failed executions. Furthermore, all of the failures that IccTA encountered were due to exceeding the alloted amount of time while most of the failures in DroidSafe were attributed to running out of memory. It should also be noted that although DroidSafe only successfully analyzed 10 of the 50 Android applications, it reported 384 information flows, which is a exceedingly high number compared to the other two tools that had a much greater number of successful executions. Many of these flows were reported from one application, Email Pop-up, which Droid- Safe reported 237 flows.

VI. Discussion

While comparing the various static analysis tools in terms of execution time, we look only at the time that it takes for the tools to finish their analysis, regardless of whether or not there was a flow detected. This is useful in regards to cost for an organization to implement a static taint analysis tool into their screening process for application submissions. The longer a tool takes, the more resources it consumes and the more cost it induces on an organization. From the results, we see that FlowDroid performs significantly quicker than the other two tools. IccTA and DroidSafe were very similar in regards to execution time, although the time taken for IccTA is much more variable than that of DroidSafe. As such, it will be much cheaper for an organization to use FlowDroid than the other two tools. As for an individual developer or a customer that wants to use one of these tools to test an application, the amount of time that it takes to execute may not be of concern. However, when looking at tools such as DroidSafe, we were unable to successfully execute the tool on 80% of the applications in the F-Droid application suite which represent realworld Android applications. If the average developer or customer does not have access to powerful enough hardware to even run the tool, it diminishes its value for potential users.

In terms of flow detection, we look not only at the raw number of flows detected but the effectiveness of the tools to accurately and precisely detect the information flows of an application. On the DROIDBENCH application suite, FlowDroid and IccTA performed very similar in successfully reported the information flows. DroidSafe on the other hand performed much better on the benchmarking suite. DroidSafe was able to detect the highest number of true flows and had the best accuracy and f-score measures. However, DroidSafe also produced the highest number of false positives. This leads us to question the high number of flows detected in the F-Droid suite by DroidSafe. There is a high likelihood that a large portion of the 384 flows detected could be false positives.

VII. Related Work

AppIntent [23] is an automated tool that aims to detect privacy leakage by creating a sequence of user interface manipulations that will initiate a sequence of events that will cause a transmission of sensitive data in an Android device. These events are then analyzed to derive the intent of the operations leading up to the sensitive data transmission and determine whether the user intended for the data to be transmitted or not. This is done by using static taint analysis to extract all possible inputs that lead to paths that could result in data transmission of a restricted resource. Using these inputs, they then automate the application execution on a step by step basis to generate user interface manipulations that will lead to a transmission of sensitive data. These results are then highlighted and displayed to a human analyst who then determines whether the transmission was indeed intended by the events that preceded it.

AndroidLeaks [12] is another static analysis tool that is designed to identify potential privacy leaks in Android applications. This tools uses data flow analysis to determine if sensitive information has reached a sink as in a program exit point that leads to the Internet. AndroidLeaks leverages WALA (Watson Libraries for Analysis) [4].

ScanDal [15] is another automated tool that attempts to detect privacy leaks in Android applications. ScanDal uses a strict static analysis approach that uses Dalvik bytecode as it's input. The benefit of this is that ScanDal can be run on any application as it doesn't need to the source code. Many other approaches must use tools such as DED [9] to decompile the Dalvik bytecode into Java source code. This present many problems as the decompiled code may not be complete and fails in many instances. ScanDal translates the Dalvik bytecode into a subset of Dalvik instructions that they called Dalvik Core, an intermediate language. The translated Dalvik Core code is then analyzed and if there is a value that is created at

an information source, it is analyzed to see if it flows out of a information sink and if so, is considered a privacy leak.

ApkCombiner [17] is an additional tool that allows for the detection of information leak that can occur between two running applications on an Android device. ApkCombiner essentially takes multiple Android applications and combines them together into a single .apk file that cant be used with other static taint analysis tools that focus on intercomponent communication to detect privacy leaks. This abstracts the inter-application communication away from existing tools and extends their functionality without alteration.

Other tools such as AsDroid [14] detect stealthy behaviors in Android applications using the using the text in the user interface comparing them to the program behavior to determine if there are contradictions with the proposed action of the application and what really happens behind the scenes. AsDroid was able to detect stealthy behaviors such as SMS messages, HTTP connections, and phone calls. While AsDroid was successful in detecting stealthy behaviors, it could not indicate the intent behind the behaviors, just that they were not consistent with the user interface.

While there is a lot of work in the realm of privacy leak and information flow detection for Android devices, there has been no prior work in comparing these tools for their efficiency and effectiveness. Our work lays ground for investigating the worth of various analysis tools and will hopefully spur more defined measures of comparison for these important tools.

VIII. Conclusion and Future Work

Privacy leak detection has steadily become a important issue in terms of mobile applications. As mobile platforms continue to be a target for malicious users seeking the private information of others, research will be necessary to detect and remove these threats. Tools such as static taint analysis tools can offer us an automated way to detect potential threats in Android applications. This paper deals with comparing some of the state of the arts tools in the area to determine their effectiveness as well as efficiency, which can play an important role when looking to implement such a tool on a large scale.

The results show that the effectiveness of a tool comes at a cost of efficiency. While DroidSafe performed the best in terms of accuracy in the detection of information flows, it suffered in terms of execution time and resource consumption. FlowDroid and IccTA, on the other hand, ran more efficiently but at the cost of missing more potential leaks in applications.

In future work, we will increase the number of applications and investigate more taint analysis tools. We will also investigate other sources for new applications suite such as Contagion Mini Dump3 and Malware DB4 that contain known Android Malware. We will also pursue Android dynamic analysis tools to compare both approaches of taint analysis on the Android platform.

References

[1] Android open source project. https://source.android.com.
[2] F-droid. https://f-droid.org/.
[3] Smartphone os market share, 2015 q2. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.
[4] Watson libraries for analysis, wala. http://wala.sourceforge.net/wiki/index.php/Main_Page.
[5] S. Arzt, S. Rasthofer, and E. Bodden. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013-0114*, 2013.
[6] 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. *Acm Sigplan Notices*, 49(6):259–269, 2014.
[7] 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*, pages 27–38. ACM, 2012.
[8] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.
[9] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX security symposium*, volume 2, page 2, 2011.
[10] 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*, pages 627–638. ACM, 2011.
[11] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the eighth symposium on usable privacy and security*, page 3. ACM, 2012.
[12] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *International Conference on Trust and Trustworthy Computing*, pages 291–307. Springer, 2012.
[13] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*. Citeseer, 2015.
[14] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1036–1046. ACM, 2014.
[15] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center. Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, 12, 2012.
[16] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.
[17] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, and Y. Le Traon. Apkcombiner: combining multiple android apps to support inter-app analysis. In *IFIP International Information Security Conference*, pages 513–527. Springer, 2015.
[18] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 280–291. IEEE Press, 2015.
[19] B. Livshits. Stanford securibench. http://suif.stanford.edu/livshits/securibench, 2005.
[20] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 77–88. IEEE Press, 2015.
[21] F. Shen, N. Vishnubhotla, C. Todarka, M. Arora, B. Dhandapani, E. J. Lehner, S. Y. Ko, and L. Ziarek. Information flows as a permission mechanism. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 515–526. ACM, 2014.
[22] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.
[23] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.