Scientific Research Publishing

Communications and Network, 2017, 9, 192-206
http://www.scirp.org/journal/cn
ISSN Online: 1947-3826
ISSN Print: 1949-2421

DroidData: Tracking and Monitoring Data Transmission in the Android Operating System

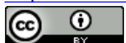
Hani Alshahrani[1,2], Abdulrahman Alzahrani[1], Alexandra Hanton[3], Ali Alshehri[1], Huirong Fu[1], Ye Zhu[4]

[1]Oakland University, Rochester, MI, USA
[2]Najran University, Najran, KSA
[3]Loyola University Chicago, Chicago, IL, USA
[4]Cleveland State University, Cleveland, OH, USA
Email: hmalshahrani@oakland.edu, aalzahrani@oakland.edu, aaalshehri@oakland.edu, fu@oakland.edu, ahanton1@luc.edu, y.zhu61@csuohio.edu

How to cite this paper: Alshahrani, H., Alzahrani, A., Hanton, A., Alshehri, A., Fu, H. and Zhu, Y. (2017) DroidData: Tracking and Monitoring Data Transmission in the Android Operating System. *Communications and Network*, **9**, 192-206.
https://doi.org/10.4236/cn.2017.93014

Received: July 29, 2017
Accepted: August 26, 2017
Published: August 29, 2017

Copyright © 2017 by authors and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).
http://creativecommons.org/licenses/by/4.0/



Open Access

Abstract

Most of the millions of Android users worldwide use applications from the official Android market (Google Play store) and unregulated alternative markets to get more functionality from their devices. Many of these applications transmit sensitive data stored on the device, either maliciously or accidentally, to outside networks. In this paper, we will study the ways that Android applications transmit data to outside servers and propose a user-friendly application, DroidData, to inform and protect the user from these security risks. We will use tools such as TaintDroid, AppIntent, and Securacy to propose an application that reveals what types of data are being transmitted from apps, the location to which the data is being transmitted, whether the data is being transmitted through a secure channel (such as HTTPS) and whether the user is aware that the information is being transmitted. The application will generate a report that allows the user to block the application that leaks sensitive information. In doing so, we will examine the importance, relevance, and prevalence of these Android Data security issues.

Keywords

Android, Security, Privacy, Tracking Data, Malware

1. Introduction

Smartphones continue to become increasingly ubiquitous, and Android leads the way in the field with over 75 percent of the market share [1]. As smartphones gain popularity and functionality, an increasing number of people use their de-

vices for privacy-sensitive activities, such as banking, health-monitoring, and business. This exposes users to privacy risks that may not be present outside of a mobile environment.

Though we know that many applications transfer privacy-sensitive information to outside servers, it is difficult to determine when the transmissions are legitimately required for the functionality of the application, versus when it constitutes a data leakage or theft. Users typically understand and accept transmissions that allow them to receive a service, such as sending location information to a maps application [2]. Many common applications also perform their computing in the cloud, which is generally an acceptable use of data when security measures are in place [3]. While this can be a useful feature, it makes it difficult to determine when data transmissions to the cloud are essential for application functionality.

Regardless of the nature of the transmission, it is almost impossible to know when data transmissions occur. Even the most technically-aware users have very little way to determine how applications are using the data on their devices, as almost 70 percent of a device's network traffic is invisible to the user [4]. Additionally, most Android applications do not provide a term of Use or End-User License Agreement, meaning even those who bother to read it do not have access to documentation that explains how the application handles sensitive information [3]. This lack of transparency makes it easy for an application to send data in ways that are not required for the functionality of the application, whether that be to steal data or to provide customized advertisements. The user has no way to know what data is being transmitted and when.

Though work has been done to create a method that tracks data transmission in Android applications, many require root access or modification of the Android source code. While these methods are effective for experts to monitor data, average users do not have the technical knowledge required to use these tools. This leaves users vulnerable to unintended data transmission. Also, existing tools use either static or dynamic analysis to search for code paths that allow data transmission. Neither of these methods is sufficient on its own to catch all privacy leaks without high false positives [5]. This indicates that existing works do not accurately report enough data transmissions to protect against data loss, especially for the average user.

In this paper, we propose a user-friendly tool that will run as a downloadable application on Android smartphones and use a combination of static and dynamic analysis to detect transmissions of privacy-sensitive data out of the device. Our contributions include:

- The novel method of combining two types of analyses, which allows DroidData to catch more data transmissions than existing works.
- A well-developed and easy-to-use user-interface that allows users to understand data transmissions and block applications that transmit their personal information inappropriately.

- More informative results about the transmissions than other tools, providing information such as the security of the data transmission.

The paper is organized as follows. Sections 2 and 3 present our problem statement and an overview of Android's architecture respectively. Section 4 provides an example of code that leaks sensitive data. Section 5 explains how static and dynamic analysis will cooperate in DroidData in order to track and monitor data. Section 6 explains how we use symbolic execution to identify which GUI events caused a data leak. Section 7 explains the related works. Section 8 mentions some limitations of our tool and how we will work on them in the future, and section 9 contains the conclusion.

2. Statement of the Problem

Studying the ways that Android applications transmit data to outside servers and propose a user-friendly application, DroidData, to inform and protect the user from these security risks.

3. Android Overview

We begin by presenting an overview of Android in order to explain how DroidData will function and protect users. Android is an open-source platform developed by Google and the Open-Handset Alliance [6]. Initially released in 2007, Android has gone through many updates and changes to improve its security and user experience. At the time of this writing, the latest version is Marshmallow. We will discuss some of the updated features in this section.

3.1. Android Architecture

Android is built in a traditional Unix paradigm [7] as shown in **Figure 1**. At the bottom of Android architecture is a modified Linux Kernel, customized with features that enhance security and improve mobile performance, in addition to the traditional drivers, process management, and file-system access [7]. An important and unique feature of the Kernel that is specific to Android is the Binder, which allows IPC/ICC (Inter-Parcel/Inter-Component Communication). This allows applications to share data and services and is crucial to the Android security model [8].

The next level above is the libraries, which include native libraries and Java Runtime Libraries. These are frequently accessed by applications via system services and provide many device functionalities [7]. Also at this layer is the Dalvik Virtual Machine. The DVM is a Java Virtual Machine designed specifically for mobile computing. It runs .dex files that are packaged in Java libraries and APK files, which are the folders used to download Android applications [7].

The next layer provides the application framework, which is responsible for distributing system services to applications and providing the crucial features of mobile devices. This layer also manages the data from device sensors and distributes it to the appropriate applications. For example, in this section we find the

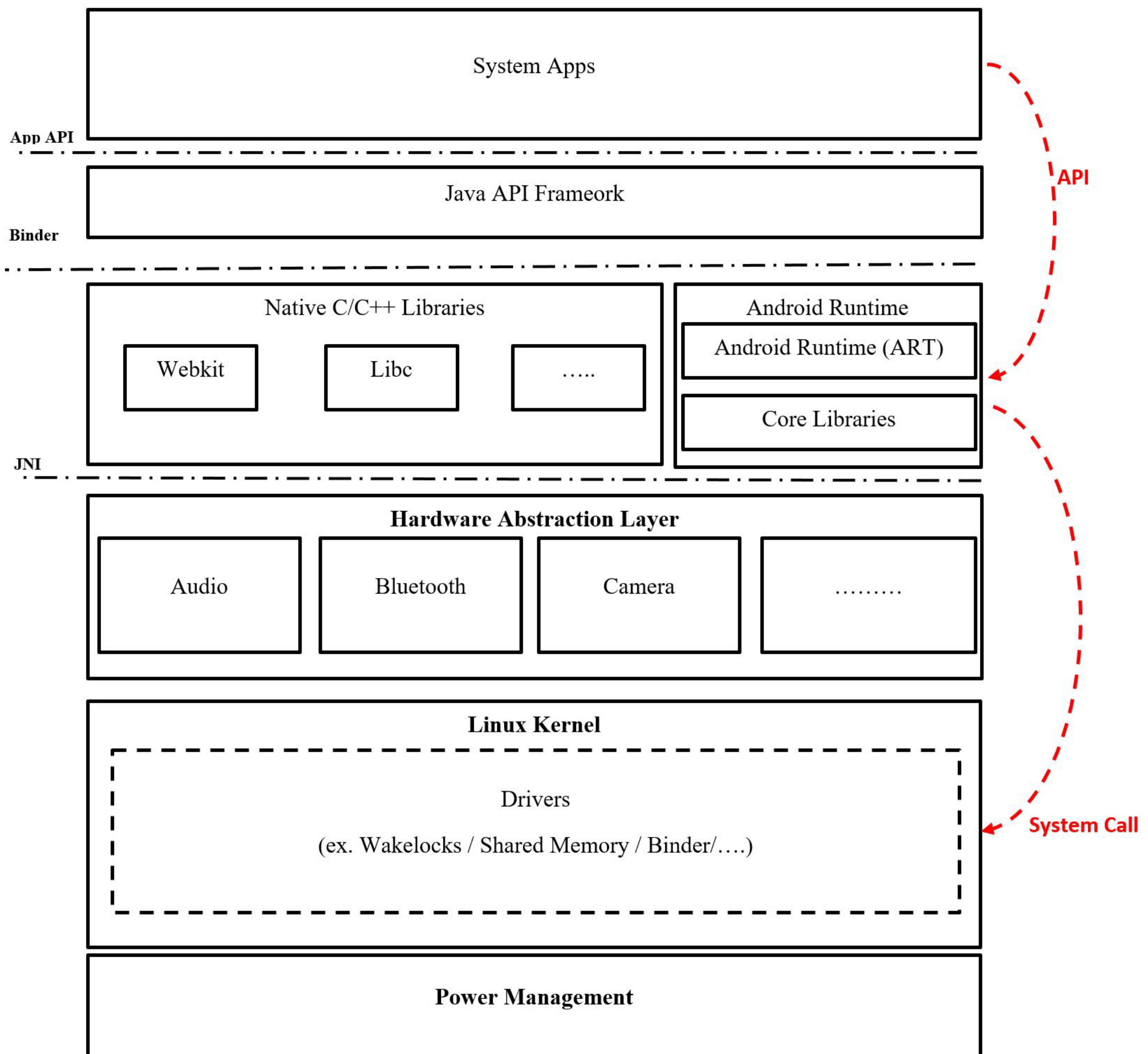


Figure 1. Android architecture.

telephony manager to provide calling abilities to the device [7]. Most of these are written in Java, like most of the Android applications they serve [7].

Android applications, which may come with the system or be downloaded by the user, are contained in an APK file, which contains a variety of items besides the source code. It also might contain XML layout files, unique libraries, the META-INF directory, which contains the signature for the package, the assets directory of application assets, and the XML manifest [8]. This manifest is one of the most important pieces of the application and contains key information, such as the name, components, permissions, and API of the application [9]. The components are key parts of the app and include activities, services, broadcast receivers, and content providers. Each activity represents a single screen in the user interface of an application. Services provide long-running functions in the

background, such as a download. Broadcast receivers receive messages from the system and other apps while the user has the application open. For example, a broadcast receiver would allow the user to receive a warning about low battery while using the app. Content providers manage and protect data within an application, sharing it where appropriate [8].

3.2. Android Security Model and Threats

Sandboxing is the central feature of the Android security model, and focuses on keeping applications separate to protect user and device data. Each application has a user ID (UID) and is given its own piece of memory that it cannot share with other applications [7]. Each application also runs in its own process, leaving it isolated, or sandboxed, from other applications [7].

When an application is downloaded, the user must then approve or disapprove a list of permissions that have been declared in the manifest [8]. These allow the application to access the Internet, device sensors or identifiers, and other services. Not all applications can request all permissions. Some are considered Signature or SignatureOrSystem, meaning that only applications with the same certificate as the declaring application can be granted the permission [8]. The latest version of Android at the time of this writing, Marshmallow, introduced fine-grained permissions management, removing the all-or-nothing permissions approval seen in earlier versions and allowing users to revoke permissions after the application had downloaded [10]. This allows the users to have more control of the access their applications have, but many users still do not take the time to consider the permissions they are granting. This is especially problematic because 70 percent of malicious applications over-request permissions [8]. As we will see in the example below, many malicious applications pose as harmless ones and fool the user into downloading them when the permissions may serve as a red flag that something is wrong. This becomes a dangerous situation that can lead to information leakage, which DroidData would prevent.

Even when users are paying attention to the permissions they grant, collusion presents a threat to their security [8]. Applications with the same certificate, usually meaning they have the same developer, can be downloaded with the same UID [7]. This allows them to run in the same process and share services and data. For example, a user might download two applications and grant one location information permission and another internet permission, not realize the two are connected, and have his or her location information leaked to an unknown server. This presents an instance in which data transmission is occurring completely unbeknownst to the user, a situation DroidData seeks to catch and prevent.

Another threat is the repackaging of applications. Malicious code, such as Trojans Gemini and KungFu, can decompile an APK file and inject it with malicious code, then disguise it as a normal application without the user's knowledge [8]. These seemingly legitimate applications can then leak personal data. Droid-

Data would also protect against this type of threat.

4. Example

We begin by giving an example of how a malicious application disguised as a normal one can transmit a user's privacy-sensitive data without his or her knowledge.

The example code in Figure 2 is taken from an actual malicious application that poses as a kitchen timer. One of its malicious activities involves stealing the user's account information, such as usernames and passwords, and transmitting them to an outside URI.

Figure 2 illustrates both sources and sinks, all of which are underlined in the code snippets. A source is the origin of privacy-sensitive information, which we see here with the method GetAccounts(). Privacy-sensitive constitutes anything that is unique to the user. It is not limited to items such as account information or location information, which are typically seen as vulnerable. For example, a less obvious source of privacy-sensitive information is device identifiers, which can be misused by malicious applications. A sink is the part of the code where sensitive data is transmitted from the device. In this case, we see that occur at Main. this. DoPost, followed by the URL of the malicious site. This shows that it

```
  Public void onReceive(Context paramContext, Intent
paramIntent)
    {
      Main.this.kitchenTimerService.schedule(300000L);
      if (Main.this.ctf.intValue() == 0)
      {
        Main.this.ctf = Integer.valueOf(1);
        paramIntent =
(TelephonyManager)Main.this.getSystemService("phone");
        Object localObject =
AccountManager.get(Main.this).getAccounts();
        paramContext = "";
        int j = localObject.length;
        int I = 0;
        for (;;)
        {
          if (I >= j)
          {
            localObject =
Main.this.doPost("http://ibbeancom.com/entry.php?id=4","");
                    paramContext = new
       Intent("android.intent.action.VIEW",
       Uri.parse("http://ibbeancom.com/send.php?a_id=" +
       paramIntent.getDeviceId() + "&telno=" +
       paramIntent.getLine1Number() + "&m_addr=" +
       paramContext + "&usr_id=" + (String)localObject));
            Main.this.startActivity(paramContext);
                  Main.this.moveTaskToBack(true);
```

Figure 2. Example of malicious app.

is easy for an application to transmit data without a user's knowledge, as what the user thinks is a kitchen timer is actually stealing their personal information.

It is fairly obvious, even to a person without an extensive background in programming, that the example above is malicious. The following code in **Figure 3** shows that it is not always easy to identify malicious code in a program.

Figure 3 comes from a malicious application that parades, ironically, as antivirus software, then routes the user to malicious web pages and encourages him or her to click on additional malicious links. This program was significantly longer and more complex than the previous kitchen timer malware, showing the importance of our tool to perform this process automatically. The code shows sources in each line of code as the program seeks to obtain information about the device, such as "device_country" and "device_android _id_hash." Not only is this code more difficult to read, but the data it is obtaining seems less sensitive. These pieces of device information still expose the user to security risks if they are leaked. DroidData will protect this vulnerable information that other tools may overlook.

Attack model: These are examples of overtly malicious applications; however, DroidData can also be used to detect accidental or benign data leaks. When looking at applications that are maliciously attempting to steal data, the tool stands up to current available malware. Most malware are able to function because of excessive permissions granted by unknowing users, and our application tracks the resulting data leaks [11]. This remains true even if the attacker changes the installation environment of the application inputs [12]. DroidData is based on the assumption that the malware operates within the Android security framework without exploiting it, which is consistent with current malware [12]. We also assume that the attacker does not use implicit flows to transmit data, which we discuss further in section VII. The first code snippet presents a scenario where these attack conditions are met. The user has granted too many permissions to an application that poses as a simple kitchen timer, but requests permissions such as network connectivity, access to device and account information, and location details. It works within the Android framework because it does request these permissions in the manifest and they are legitimately granted

```
          JSONObject localJSONObject = new JSONObject();
          localJSONObject.put("av",
paramgq.getString(paramgq.getColumnIndexOrThrow("app_ver
sion")));
          localJSONObject.put("dac",
paramgq.getString(paramgq.getColumnIndexOrThrow("network
_type")));
          localJSONObject.put("du",
paramgq.getString(paramgq.getColumnIndexOrThrow("device_
android_id_hash")));
          localJSONObject.put("dc",
paramgq.getString(paramgq.getColumnIndexOrThrow("device_
country")));
```

Figure 3. Example of malicious app.

by the user. It exploits the user, but not the Android security model.

5. Implementation

DroidData will use both static and dynamic analysis to track and monitor data transmissions. In order to accomplish this type of analysis, our application will sit below the application frameworks and above the libraries, Dalvik Virtual Machine, and core libraries, as illustrated in **Figure 4**. This is similar to the position that a traditional desktop antivirus has. This will provide our application with the access to the ICC components, device sensors and resources it needs to accurately track data as it moves through the system. Static and dynamic analysis will cooperate by each generating a separate report when a threat is found. In this way, the two will not necessarily work together; rather, they will each look for data leaks independently to increase security by ensuring that no potential leaks are missed.

Taint analysis is a widely accepted method for tracking data transmission in the Android OS. It can be performed either dynamically or statically, and both our static and dynamic analysis will use taint tracking [13]. Taint analysis involves tagging data of interest at sources, in this case privacy-sensitive data, with a "taint tag" and tracking the propagation of the taint through the rest of the code [3]. Taint tags are stored in adjacent memory and logs are used to determine when the tainted data leaves the system [14]. Taint propagation must be performed carefully in order to prevent taint explosion, when almost everything in the system accidentally becomes tainted [3].

5.1. Static Analysis

The first step our application will perform in analyzing applications is static analysis, as illustrated in **Figure 5**. This provides the best code coverage because it examines all possible data paths, enabling it to detect around 85 percent of flaws in the code [15]. It is also safer because potentially malicious code

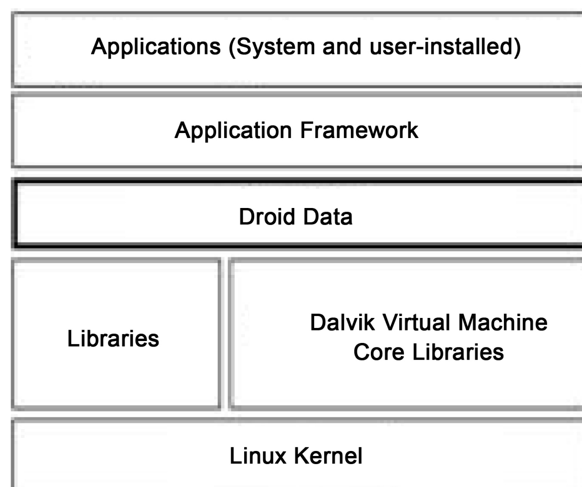


Figure 4. DroidData implementation.

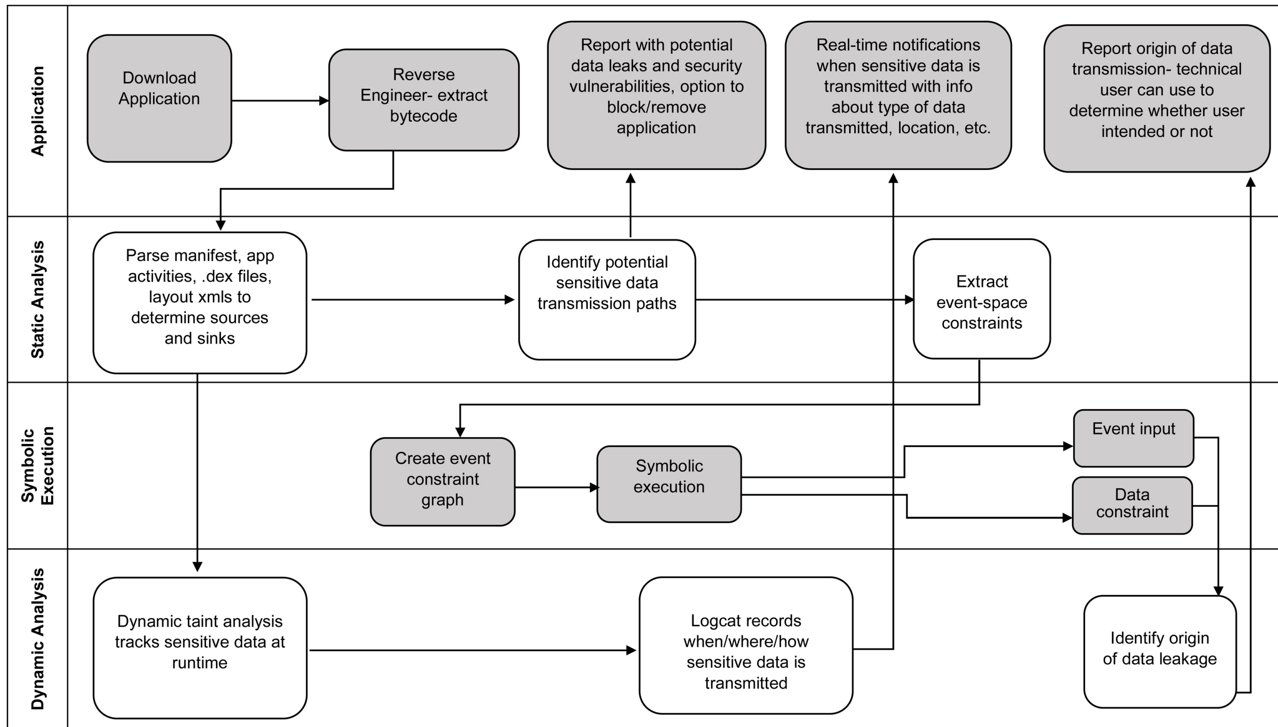


Figure 5. Data droid workflow.

does not actually execute [5]. For this reason, it was crucial that we include static analysis in our tool, so that users did not lose sensitive information, then be notified when it was already too late. Static analysis begins by converting the Dalvik bytecode from an APK file back into Java code so that it can be analyzed [12]. We did this manually with tools such as JD-GUI, apktool, and dex-2-jar [16] [17] [18]. We plan to automate this process in DroidData.

Once we have extracted the bytecode, we must analyze it in order to identify the sources and sinks. This must be comprehensive in order to identify all possible places of origin and loss of sensitive data. To do make it so, we must look not only at the Java classes, but also at the XML manifest and layouts, any files and libraries associated with the application, and any other items that may be in the .apk file [12]. We will implement a modified version of the open-source tool SuSi, which uses supervised machine learning to generate a categorized list of sources and sinks in an Android application [19]. Using the sources and sinks that we identify, we can identify all of the potential paths that connect sources to sinks using static taint analysis, which we discussed earlier. These potential paths are then reported to the user in a report that details the types of data that may be transmitted and other known information about the paths.

There are some potential flaws in static analysis, and these must be noted to the user in the initial report. It will be made clear that these are only potential transmissions, and that use of the application does not guarantee that the specific code path will ever be executed and data transmitted as such [5]. It may also be the case that not all information about the transmission will be known

through static analysis. It is difficult to tell what type of channel the data will be transmitted through via static analysis, and code paths that interact with other parts of the system, such as libraries, may not be uncovered [5]. These are all reasons that we also implement dynamic analysis, but we will make clear to the user in this initial report that the results of static analysis may not reflect actual data transmissions that will occur. As an initial method of analysis, we went through the code of known malicious applications to determine potential sources and sinks of sensitive information. This is the same process that will be performed automatically in our application during static analysis. We found a wide variety of sources and sinks during our manual analysis, including several instances of location and account information being sent to outside sources.

Our application will also run an additional type of static analysis, the extraction of event-space constraints. We will discuss this in the next section where we explain symbolic execution.

5.2. Dynamic Analysis

DroidData begins to perform dynamic analysis as soon as the subject application is run, again illustrated in **Figure 5**. Running dynamic analysis addresses many of the concerns present when only running static analysis and provides the user feedback about the way the application uses data at runtime. This allows the user to know how the application actually uses data, not just how it might use data. Dynamic analysis also better shows how the application interacts with other files and libraries, and may detect some code paths that are too complicated for static analysis to catch [5]. As a downfall, it can be avoided by truly malicious applications [5]. This makes it important that we are using it in combination with static analysis.

When dynamic taint analysis runs, we notify the user of each transmission that it finds. This notification will include the type of data transmitted, the IP address and approximate geographic location of the server to which it was sent, and whether the data was sent through a secure channel, such as HTTPS. This will give the user another opportunity to block the application if data has been transmitted inappropriately. The application also dynamically analyzes the results of symbolic execution, which we discuss next.

6. Determining User Intention

Similar to the related work AppIntent [20], DroidData will provide the sequence of GUI manipulations that lead to the data transmission in order to determine the origin of the leak. A close-up of our implementation of this process is shown in **Figure 5**. The origins and paths of data transmissions are determined using guided symbolic execution, a process pioneered by AppIntent [20]. This starts by extracting the event-space constraints during static analysis [20]. To do this, we must identify the critical and essential events, where critical events are those necessary for the transmission and essential events are prerequisites to critical

events [20]. These are then used to create the event constraint graph, which shows all possible data transmission paths and the GUI events related to each one [20]. It is made carefully so as to not violate the Android lifecycle, which is the set up methods that are used in Android instead of a main method to start and stop the application [21]. Symbolic execution then runs and is used to determine the possible event sequences that lead to data transmission [20]. This involves narrowing the search space of event sequences by traversing the graph to find the minimal chains of events that lead to a transmission [20]. The result is the event inputs and data constraints that must be present to transmit the data, thus providing the preconditions of data transmission [20]. These are not in an easily understandable form, so a dynamic analysis platform identifies which function of the application is used when each GUI manipulation occurs to determine which caused the transmission.

AppIntent uses a human analyst to look at the results of the dynamic analysis and determine whether or not a transmission was user-intended or not, which is not something we are able to implement in a downloadable mobile application [20]. For this reason, DroidData will provide the sequence of events that lead to the data transmission so that a technologically-aware user can use them to determine whether the transmission was intentional. For users who may not be able to understand the sequence of GUI manipulations, the static and dynamic taint analysis and their resulting reports will provide a clear enough picture for them to understand the transmission, even if they do not know whether it was intentional or not.

7. Related Works

A variety of tools have used taint analysis to analyze Android applications for data loss are listed in **Table 1**.

TaintDroid [3] uses dynamic taint analysis to track sensitive data as it moves through the Android operating system and provides the user with a notification when data leaves the system. It uses dynamic taint analysis to track sensitive data as it moves through the Android operating system and provides the user with a notification when data leaves the system. Widely used in the field, it tracks taint

Table 1. Applications comparison.

Applications	Features				
	Static Analysis	*Dynamic Analysis*	*Creates Report*	*Requires Tech Knowledge*	*User Can Block App*
DroidData	X	X	X		X
Taint-Droid [3]		X		X	
Securacy [22]		X	X	X	
Flow-Droid [12]	X		X	X	
AppIntent [20]	X	X		X	
DroidScope [13]		X		X	

at five levels in the device: variable-level, message-level, file-level, and method-level. A disadvantage of TaintDroid is that it requires root access and modifications to the Android source code, which is difficult for a non-technical user. DroidData would be downloaded like a regular application and provide more information for users without a technical background.

Securacy [22] uses TaintDroid's functionality and combines it with a crowd-sourcing tool where users can share information about security issues they experienced with an application and provide a rating for other users to consider before downloading the application. This has the disadvantage that according to their own investigation, only five out of one hundred of their participants rated an app. Rating an application is often considered inconvenient by users and is not always accurate, based on varying opinions and understandings of privacy by different users. In DroidData, we prefer to provide the user with a comprehensive report of our own creation that gives an accurate description of the type of data being sent out and the location to which it is sent, instead of relying on other to share their experiences with the application.

FlowDroid [12] performs static taint analysis after analyzing an application's manifest, XML files, and .dex files to generate a "dummy" main method that considers Android lifecycle and callback methods in determining the potential data paths. This context, flow, field and object sensitivity increases precision. Like other taint analysis tools, FlowDroid falls short by using only static analysis. In addition, when testing FlowDroid ourselves, we had a difficult time understanding the results of the analysis. For a non-technical user, using the tool and understanding the results would be impossible.

AppIntent [20], as discussed earlier in the paper, uses symbolic execution to determine the GUI manipulations that lead up to a data transmission to allow an expert to determine whether is intended by the user or not. This draws a distinction between true data leaks and transmissions that are necessary for application functionality. It pioneers a technique called "Event-space Constraint Guided Symbolic Execution" that identifies all possible execution paths using static analysis, then identifies critical events to the transmission using an event constraint graph. It then uses dynamic analysis to determine what UI manipulations led to the data transmission. While this tool is unique in seeking to determine whether data transmission is a privacy leak, it requires that a human analyst be presented with the results of the dynamic analysis to determine user intention, and they note that it is probably impossible for that process to be completely automated.

DroidScope [13] takes their dynamic analysis tool off-device by virtualizing the device's semantics and mirroring the Android hardware, operating system, and virtual machine. It also provides analysis tools, one of which is a taint tracker that detects data leakage. The virtualization prevents malicious applications from evading the anti-malware tools, which increases user safety. The emulated environment, however, has many important differences from an actual device,

such as sensors that are difficult to emulate. This makes it difficult to protect all sensitive data.

8. Future Work

We are currently in the process of implementing DroidData as it is explained in this paper. We will continue to test and improve it moving forward. In the future, we would like to continue to deal with some of the following issues that exist in the proposed version. DroidData, like most other current analysis tools for Android, is unable to track implicit, or control, data flows. Some malicious code uses implicit flows to exploit security mechanisms and avoid detection, so this is something to protect against in the future [23]. Tools such as Flow Caml support implicit data flow monitoring in specific languages, and we are interested in exploring such an implementation in a mobile software in the future [24].

The symbolic execution features of our tool that are meant to determine the origin of data leaks are of little use to users who are not technologically knowledgeable. The results of the dynamic analysis produce the event that was the origin of the data leak, but it is difficult for someone without an understanding of the steps of the process to interpret that information and determine whether the data transmission was intentional or not. A potential future work is to put that GUI event into a description that the average user can understand in order to be better able to decide whether the transmission was something he or she requested.

9. Conclusion

We have proposed DroidData, our novel tool that uses both static and dynamic analysis to track and monitor data transmission in Android applications. This approach minimizes false positives and increases code coverage to catch the maximum number of data leaks. We also implement symbolic execution in order to determine the origin of the data leak, and we provide a clear user interface that allows users without a technological background to understand where applications send their data and whether it is being sent through secure channels, and the opportunity to block those that transmit sensitive information inappropriately.

Acknowledgements

This work was supported in part by NSF under grants CNS-1460897, DGE-1623713. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M.S., Conti, M. and Rajarajan, M. (2015) Android Security: A Survey of Issues, Malware Penetration, and De-

fenses. *IEEE Communications Surveys & Tutorials*, **17**, 998-1022.
http://ieeexplore.ieee.org/abstract/document/6999911/?section=abstract
https://doi.org/10.1109/COMST.2014.2386139

[2] Gibler, C., Crussell, J., Erickson, J. and Chen, H. (2012) Androidleaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. *Proceedings of the* 5*th International Conference on Trust and Trustworthy Computing*, Vienna, 13-15 June 2012, 291-307. https://doi.org/10.1007/978-3-642-30921-2_17

[3] Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P. and Sheth, A.N. (2010) Taint Droid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *Proceedings of the* 9*th USENIX Conference on Operating Systems Design and Implementation*, Vancouver, 4-6 October 2010, 393-407.

[4] Tu, G.H., Peng, C.Y., Li, C.Y., Ma, X.Y., Wang, H.Y., Wang, T. and Lu, S.W. (2013) Accounting for Roaming Users on Mobile Data Access: Issues and Root Causes. *Proceeding of the* 11*th Annual International Conference on Mobile Systems, Applications, and Services*, Taipei, 25-28 June 2013, 305-318.
https://doi.org/10.1145/2462456.2464439

[5] Institute for System Programming of the Russian Academy of Sciences (2016) Static Analysis vs. Dynamic Analysis. http://linuxtesting.org/static-vs-dynamic

[6] Alliance FAQ (2017) http://www.openhandsetalliance.com/oha_faq.html

[7] Elenkov, N. (2014) Android Security Internals: An In-Depth Guide to Android's Security Architecture. No Starch Press, San Francisco.

[8] Rashidi, B. and Fung, C. (2015) A Survey of Android Security Threats and Defenses. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, **6**, 2015.

[9] Android Developer (2016) App Manifest.
https://developer.android.com/guide/topics/manifest/manifest-intro.html

[10] Carlon, K. (2016) Android 6.0 Marshmallow: All the Key Features Explained.
https://www.androidpit.com/android-m-release-date-news-features-name

[11] Bartel, A., Klein, J., Le Traon, Y. and Monperrus, M. (2012) Automatically Securing Permission-Based Software by Reducing the Attack Surface: An Application to Android. *Proceedings of the* 27*th IEEE/ACM International Conference on Automated Software Engineering*, Essen, 3-7 September 2012, 274-277.
https://doi.org/10.1145/2351676.2351722

[12] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D. and McDaniel, P. (2014) Flowdroid: Precise Context, Ow, Eld, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. *SIGPLAN Notices*, **49**, 259-269.
https://doi.org/10.1145/2666356.2594299

[13] Kwong, L.Y. and Yin, H. (2012) Droidscope: Seamlessly Reconstructing the Os and Dalvik Semantic Views for Dynamic Android Malware Analysis. *Proceedings of the* 21*st USENIX Conference on Security Symposium*, Bellevue, 8-10 August 2012, 29.

[14] McClurg, J., Friedman, J. and Ng, W. (2013) Android Privacy Leak Detection via Dynamic Taint Analysis. Northwestern University, Evanston, IL.
http://www.jrmcclurg.com/papers/internet_security_final_report.pdf

[15] Brain, R. (2010) Dynamic Code Analysis vs. Static Analysis Source Code Testing.
http://www.computerweekly.com/answer/Dynamic-code-analysis-vs-static-analysis-source-code-testing

[16] Java Decompiler (2017) Download. http://jd.benow.ca/

[17] Apktool (2016) A Tool for Reverse Engineering Android Apk Files.

https://ibotpeaches.github.io/Apktool/

[18] SourceForge (2016) Dex2jar. https://sourceforge.net/projects/dex2jar

[19] Arzt, S., Rasthofer, S. and Bodden, E. (2013) Susi: A Tool for the Fully Automated Classi Cation and Categorization of Android Sources and Sinks. University of Darmstadt, Darmstadt.

[20] Yang, Z.M., Yang, M., Zhang, Y., Gu, G.F., Ning, P. and Wang, X.S. (2013) Appintent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. *Proceedings of the* 2013 *ACM SIGSAC Conference on Computer* & *Communications Security*, Berlin, 4-8 November 2013, 1043-1054. https://doi.org/10.1145/2508859.2516676

[21] Android Developer (2016) Managing the Activity Lifecycle. https://developer.android.com/training/basics/activity-lifecycle/index.html

[22] Ferreira, D., Kostakos, V., Beresford, A.R., Lindqvist, J. and Dey, A.K. (2015) Securacy: An Empirical Investigation of Android Applications' Network Usage, Privacy and Security. *Proceedings of the* 8*th ACM Conference on Security* & *Privacy in Wireless and Mobile Networks*, New York, 22-26 June 2015, 1-11. https://doi.org/10.1145/2766498.2766506

[23] Russo, A., Sabelfeld, A. and Li, K. (2009) Implicit Flows in Malicious and Nonmalicious Code. *Proceedings of the* 2009 *Marktoberdorf Summer School*, Garching, 4-16 August 2009, 301-322.

[24] Simonet, V. (2015) Flow Caml. http://www.normalesup.org/~simonet/soft/flowcaml

Submit or recommend next manuscript to SCIRP and we will provide best service for you:

Accepting pre-submission inquiries through Email, Facebook, LinkedIn, Twitter, etc.
A wide selection of journals (inclusive of 9 subjects, more than 200 journals)
Providing 24-hour high-quality service
User-friendly online submission system
Fair and swift peer-review system
Efficient typesetting and proofreading procedure
Display of the result of downloads and visits, as well as the number of cited articles
Maximum dissemination of your research work

Submit your manuscript at: http://papersubmission.scirp.org/
Or contact cn@scirp.org