Taint Analysis on Gboard

Linda Nguyen
New York University

Dennis Sun
New York University

Abstract

Third party keyboard applications are popular among Android users as a way to customize their smartphone. Keyboard applications such as Google Keyboard request permissions without providing users with transparency of how they use private data. In this paper, we use TaintDroid to replicate their experimental setup with a different version of Android on a different device and analyze one application. By using TaintDroid with a different setup, we verify its reproducibility and check Google Keyboard for potential privacy violations. TaintDroid Notify 4.1.1 application analysis is unreproducible on the Nexus 4 with Android 4.3 due to a linker issue that Google claimed has been fixed, which is caused by TaintDroid's implementation of taint propagation. Future experiments should follow the exact setup of TaintDroid as described in their paper.

1 Introduction

Third party keyboard applications have been available for Android before they were available to iOS users [42]. Android users often choose to install third party keyboards, which offer more functionality than the default keyboard installed. Gboard has often been crowned the best keyboard due to the features like integrated web search, Google Translate, and gif search. Other features include gesture typing, multiple languages, predictive text, and voice-to-text [36].

When users install Google Keyboard, they are prompted to grant the application with access to:

- read your own contact card
- find accounts on the device
- read your contacts
- modify or delete the contents of your USB storage
- download files without notification
- full network access
- read terms you added to the [user-defined] dictionary
- view network connections

Users apply Gboard features with the risk of exposing their private data because they do not know where their data is distributed. iOS placed greater security restrictions on keyboards than Android, which led to later releases in keyboard applications [42].

Google's privacy statement regarding Gboard states that search results and usage statistics are sent to Google's web servers. Anything else that is typed other than your searches is not sent, including saved words on your device [17].

The privacy statement is not a catch-all. In 2012, Google claimed that users could rely on Safari's privacy settings to prevent tracking. Google was found to be bypassing Safari's tracking detection and tracking its users. Cookies on Safari are blocked with the exception of cookies for forms. Google then added an invisible form to its ads in order to install tracking cookies. When a cookie was about to expire, another one was created. Thus, Google could track users indefinitely on Safari [1].

Past informal analysis of Google Keyboard has included analyzing the installation permission request or reading Google's privacy statement [43]. The results of running network packet sniffer on iOS GBoard were inconclusive because all outbound packets were encrypted. However, no packets were sent outbound while a user was typing [11].

In this project, we will reproduce TaintDroid's setup on Gboard. The TaintDroid experiment pulled a randomly selected set of thirty applications from an early 2015 survey of the 50 most popular free applications in each category. A custom ROM with TaintDroid as an application was developed for Android 2.1. The applications were then downloaded onto the Nexus One with an inactive SIM card, using only the WiFi interface. Taint-

Droid then sent notifications as the functionality of the application was exercised. The results were then verified with tcpdump, a package analyzer. The experiment was repeated with multiple Nexus One smartphones with the same setup. We will build the provided ROM for Android 4.3 on the Nexus 4, install Gboard, and use TaintDroid Notify to record Gboard behavior.

2 Related Work

The preservation of information security by searching for leaks in sensitive information has been a point of interest in industry and in research. Examples of such tools include Privacy Oracle [21] and TightLip [41], which monitor applications like a black box. A formal description of dynamic taint analysis (DTA) was described in a 2010 paper, while detailing concerns on DTAs effectiveness like the prevalence of false positives and negatives. Several implementations have been developed, including TaintCheck and Dytan. TaintCheck detects most exploit attempts during runtime without false positives [24]. Other implementations like Dytan work with a wide range of x86 binaries and can detect web exploits [7].

Android has also been a point of interest in terms of security, especially in terms of user privacy. In order to protect user privacy, AppFence will substitute sensitive data with shadow data, and keep sensitive data from leaking out through the network when it is intended to be used only in the device [20]. A utility called Haystack was also developed to act as a measurement platform for Androids network traffic, which is applicable for investigating privacy risks on Android [27].

In 2015, researchers found a SwiftKey vulnerability that could allow remote code execution [37]. This led to a security analysis of third party Android keyboards [6]. Many applications were found to contain ideal permissions for keyloggers such as external storage or a network connection. An analysis on 125 keyboard applications showed that over 70% of applications requested one of these permissions, as well as access to other data on the phone [23].

Using only the INTERNET permission, a keyboard application could stealthily send user data to a keylogging server in a background thread. The application passed the Google Play application review process and AndroTotal, an antivirus scanner, was unable to detect the threat. A proposed solution to identifying keyloggers would be information flow tracking. Monitoring all information flow operations would cause runtime overhead, so static analysis was recommended over dynamic analysis.

TaintDroid is among one of the first implementations of DTA for Androids system and its applications. Since then, other implementations of DTA for Android have been created as attempts to improve TaintDroid. As TaintDroid was developed for older versions of Android, it is not compatible with newer Android versions with the newer Android RunTime (ART). TaintMan is a recent implementation of DTA for Android thats compatible with the new ART [40]. For detecting malware on Android systems, AndroTaint was developed to perform DTA [32].

TaintDroid is an information flow tracking tool that can track different sources of data [9]. It is the most complete dynamic analysis tool for Android [30]. TaintDroid was used to discover that 50% of the thirty tested applications sent users' locations to remote advertising servers. However, TaintDroid inherits all the issues of dynamic analysis. In fact, AntiTaintDroid is an application to bypass TaintDroid [28].

AntiTaintDroid can prevent taint propagation by avoiding direct assignments and manipulate benign code into leaking sensitive data [28]. Most dynamic taint-checkers also overlook side channel attacks. For example, keyloggers can determine keystrokes from Androids accelerometer data [4]. Worse, a malicious application could test for analysis and refrain from malicious activity [13]. TaintDroid faces also disadvantages of tracking only data flow, ignoring control flow.

Proposals to circumvent anti-taint-analysis are ineffective [5]. For example, one proposal suggests classifying all input as tainted, but that would generate too many false positives. Other defenses may include restrictions on API or memory, which may be too easily overcome. Any attempt at mitigation can lead to a higher rate of false positives that make DTA useless. Reducing mitigation of anti-taint-analysis will make DTA easier to trick and bypass, but more precise [5, 28]. The fact that DTA can be avoided may be an important factor to consider. With the number of reputable engineers and researchers, it is not out of the question that Google could develop an application resilient from analysis methods.

Although less complete, other frameworks have been developed to counter TaintDroid. The framework was not bound to the platform and analyzed on an application level [30]. Despite a 100% detection rate against DroidBench and a 91.7% coverage of TaintDroids data flows, it did not overcome any of TaintDroids issues [12].

DTA is ineffective against a stealthy attacker. Using a static analysis tool such as FlowDroid would avoid dynamic analysis challenges. FlowDroid is a precise static taint analysis tool that handles callbacks, while context, flow, field and object sensitivity. FlowDroid has more extensive code coverage and reduces false positives. Static analysis of Android applications causes discontinuities in the CFG due to inter-component communication which resolves at runtime. FlowDroid-IccTA was extended

from FlowDroid to find privacy leaks within multiple components instead of single components [22]. IccTA enables a data-flow analysis between components and detects inter-component communication privacy leaks. IccTA was able to detect 130 leaks in 12 applications from three thousand applications.

3 Background

3.1 Android Applications

Like any Java program, Android applications are first compiled by converting source code into Oracle JVM Java byte-code `.class` files. Android uses the Dalvik VM, not the Oracle JVM, so `.class` files and other `.jar` files are converted into a single file called `classes.dex`, which contains Dalvik byte-code. In Android, the Dalvik byte-codes are interpreted with the Dalvik VM interpreter. Between the `.class` files and `.jar` files, references that are used are linked by the linker to the libraries and other implementations containing the needed classes and methods. The `classes.dex` file is then combined with any other application resources, like images and layouts, into the `.apk` file that represents the Android application. These `.apk` files are distributed and installed onto the Android operating system [19]. This entire process is summarized and compared to Oracle Java compilation in Figure 1.

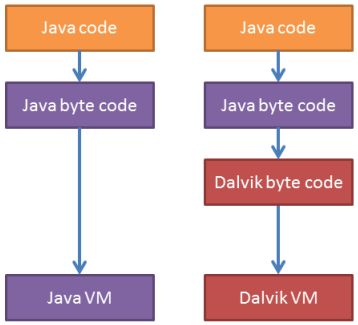

Figure 1: Oracle Java compilation and Android application compilation on the left and right, respectively.
Photo credit: [8]

3.2 Dynamic Taint Analysis

Taint analysis is the tracking of certain input throughout the execution of a program. Dynamic taint analysis (DTA) is specifically done during the execution of the program. Specific data is first marked with taint markers, and the taint markers are propagated as the tainted data propagates throughout the program. Generally, data thats based on direct reads from tainted data will also be marked with the same taint marker. This provides insight on where tainted data will reach in the program for a specific program execution, under certain conditions [31].

3.3 Real-World Application Permissions

Recent updates to the permissions to some applications on the Android Market led to concerns on user privacy, such as keyboard applications like Gboard and SwiftKey. In this instance, these keyboard applications use network access. As one would expect, the use of these permissions can be alarming or concerning to users, as their use of the network connection is rather suspicious in terms of preserving user privacy [25]. Since there is concern regarding the permissions that Gboard uses, we can hypothesize how Gboard utilizes these permissions to process user data and determine how Google may use this user data.

3.4 TaintDroid

TaintDroid is a DTA system that integrates itself into the Android operating system, tracking and analyzing the use of sensitive data by third-party Android applications. If TaintDroid detects a potential instance of misuse of sensitive data by a third-party application, TaintDroid will notify the user about the matter via a notification.

Since TaintDroid is highly coupled with the Android operating system and integrated within the system, the use of TaintDroid requires its own build of Android. Any viable builds of TaintDroid must be built with an Android build of version Jelly Bean, or earlier.

3.4.1 Taint Tag Storage

More specifically, TaintDroid taints the sensitive information and stores the taint markers in a virtual taint map. The taint tags are propagated as a trusted applications are using the sensitive information. Whenever the sensitive information is obtained by an untrusted application through interprocess communication, any values read from the tainted values receive the taint markings in the external application. Similarly to the trusted application, the taint markings are propagated with the virtual taint map in the untrusted application. If the untrusted application ever invokes a taint sink involving a tainted value, the user is notified of this event [9].

3.4.2 Interpreted Code Taint Propagation

TaintDroid utilizes taint markings that represent variables with types of uses. These consist of variables corresponding to virtual registers, class fields, instance fields, static fields, and arrays. With taint markings applied to

arrays, taint propagation must also account for object references as well as object values [9].

3.4.3 Native Code Taint Propagation

Since TaintDroid does not monitor native code, it uses manual instrumentation, heuristics, and method profiles to provide taint propagation. In particular, TaintDroid utilizes internal VM methods and JNI methods in order to achieve this [9]. TaintDroid patched the call bridge, which parses arguments and assigns a return value, in order update the taint propagation values.

3.5 Gboard

3.5.1 Glide Typing and Text Prediction

Gboard features glide typing as a form of input, which is subject to user error that may lead to Gboard interpreting input into an incorrect word. This user error can be due to physical inaccuracy or misspellings of the intended word. In order to quickly predict the user's intended word, Google developed spacial models to interpret these swipes more accurately, as an upgrade to a previous Gaussian error correction model.

Gboard utilizes localized machine learning to train a local model of the user's personalized glide typing interpretation. Using user feedback in the form of backspaces after an inaccurate autocorrect, the neural spatial model relies on a high volume of training data from Gboard's use. As of spring 2017, Google has used a year's worth of training data to improve their accuracy and speed for the text prediction and word correction in glide typing [16].

As many keyboard applications do, Gboard also provides text prediction, which requires the processing and interpretation of users' input keys and words. To map keystroke sequences to words, Gboard makes use of graphs called finite-state transducers (FST). An FST, depicted in Figure 2, represents nodes as states where a certain key sequence has already been typed, with directed edges leading from node to node representing a single keystroke along with a predicted word, based on the current keystroke sequence and the newly typed key. To predict words based on previous words, Gboard makes use of probabilistic n-gram transducers, which makes predictions of a word based on the previous $n-1$ words typed. Similarly, the n-gram transducers are graphs with nodes representing states with word sequences and directed edges representing word predictions with an associated probability [16].

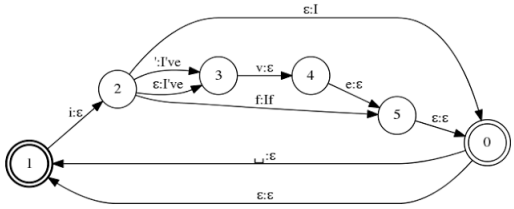

Figure 2: A small finite-state transducer that shows the states representing already-typed key sequences. The edges represent a typed key and an associated word prediction, with ε representing the empty state. The double border nodes, states 1 and 0, represent the start and end states. Photo credit: [16]

One of the advantages of FSTs is that they can be combined in a space-efficient manner. When multiple users have their own FSTs for text prediction, it is not out of the question to think that Google may combine users' FSTs to one large FST, so that it can be redistributed back to Gboard users. So not only does analyzing an FST provide insight to how Gboard process user input, but can also itself be a point of investigation surrounding the user privacy of Gboard.

3.5.2 Machine Learning in Gboard

With the use of both neural spatial models and finite state transducers, Gboard provides glide typing and text prediction as general functionalities in the application. Since autocorrected glide typing and text prediction require a large volume of training data, Google has certainly sent users' input data to a centralized cloud location to simultaneously train all users' Gboard applications. Google accomplishes this through Federated Learning, where Gboard tracks an individual user's input, generates a summary on the user's differences from the current model, and sends only the summary to Google's cloud to update its model. This is supposed to provide more accuracy in the consensus model and more immediate changes to the personalized models in individual users' applications.

Federated Learning is also supposed to provide Gboard users with more privacy, since not all of the individual training data is supposed to be sent, but just the summary. Still, this leads to the question of what form the summaries of training data are, how Gboard generates the summaries, whether there is extraneous data that can be misused, and how Google might use these summaries in ways unrelated to updating the glide typing model. Taint analysis research surrounding this aspect of Gboard should provide insight on how glide typing input is processed to generate these summaries [15].

As we know, Gboard requests for concerning permissions, like write permissions to USB storage, the ability to download files without notification, full network access, and access to the user-defined dictionary. Based

on how Google claims to use Federated Learning, we can hypothesize that Gboard uses its ability to download files without notification to update users' individual models for glide typing corrections and text prediction, and utilizes full network access to exchange summaries of training data and consensus model updates. Write permissions to USB storage may be unnecessary, so we will analyze where processed data and models are stored. Lastly, access to the user-defined dictionary may be for text predictions based on what the user adds, as well as to add its own updates based on the user's Gboard usage and model updates from other users' Gboard usage. However, the confirmation of this depends on the format and contents of the summaries of training data from the device to Google's servers.

4 Experiment Setup

4.1 Building Android

The first step is to build Android from source. TaintDroid only supports Android 2.1, Android 2.3, Android 4.1, and Android 4.3. Our project is on Android 4.3. We began with a fresh server install image of Ubuntu 14.04 LTS (Trusty Tahr). To build Android, the hardware requirements are a 64 bit environment for ¿ Android 2.3.x and at least 250 GB of free hard disk space.

To install the correct version of Java, we first removed all versions of Java.

```
$ sudo apt-get purge openjdk-\* icedtea-\*
  icedtea6-\*
```

We created an Oracle account and downloaded the binary to install JDK 6 for Android 4.3.

```
$ wget http://server-hosting-the-binary/
  jdk-6u31-linux-x64.bin
$ chmod +x jdk-6u31-linux-x64.bin
$ mv jdk1.6.0_31/ java-6-oracle
$ sudo mkdir /usr/lib/jvm
$ sudo mv java-6-oracle /usr/lib/jvm
$ java -v
```

We then install the following packages to setup the build environment.

```
$ sudo apt-get install build-essential git
  ccache automake lzop bison gperf
  build-essential zip curl
  zlib1g-dev zlib1g-dev:i386 g++-multilib
  python-networkx libxml2-utils bzip2
  libbz2-dev libbz2-1.0 libghc-bzlib-dev
  squashfs-tools pngcrush schedtool
  dpkg-dev liblz4-tool make optipng flex
  libswitch-perl
```

From this point onwards, follow the official source.android instructions.

Once the build finishes successfully, we installed `ubuntu-desktop` from `tasksel` in order to verify that the emulator runs and that there are no build environment errors for Android.

```
$ sudo apt-get install tasksel
$ sudo tasksel
```

Select Ubuntu Desktop.

We also need to VNC into the server to verify that the emulator runs, so we installed tightVNC following these instructions from DigitalOcean [10].

4.2 Building TaintDroid

We followed the download instructions from TaintDroid's download page, up until the last repo sync. We needed to then add the `--force-sync` flag to allow overwrites for all packages.

We next downloaded the proprietary driver binaries for the Nexus 4 (mako) following the instructions from the official Android developer site [14]. We then extracted the binaries into the source directory.

We finally built TaintDroid as specified in Step 4 of the TaintDroid download page. The command for lunch that we used is `lunch full_mako-eng` because we are building the ROM for the Nexus 4.

4.3 Flash the Device

We installed fastboot and adb via brew onto our local computers.

```
$ brew cask install android-platform-tools
```

Now we are able to flash the device as specified in Step 5 of the TaintDroid download page.

4.4 Using the Device

Upon using the device, we found three issues. The first was that the device could not enable WiFi. The second was that TaintDroid Notify, the application to log the behavior of Google Keyboard, seemed to be inactive. The third was that any additionally installed application would crash.

5 Challenges

5.1 Setting up the Build Environment

TaintDroid referred to the official Android website for instructions on building from source. However, the list

of packages that needed to be installed for the build environment were missing a few details. Links to the correct version of the JDK (for Android Jelly Bean) led to the wrong page. JDK 6 and JDK 7 are archived and only available for Oracle support customers. We created an account and then installed JDK 6 from the provided binary.

Instructions on creating the build environment did not include a few necessary packages, which were `lzop`, `zlib1g-dev:i386`, `bzip2`, `libbz2-dev`, `libbz2-1.0`, `libghc-bzlib-dev`, `squashfs-tools`, `pngcrush`, `schedtool`, `dpkg-dev`, `liblz4-tool`, `make`, `optipng`, `libswitch-perl`. We discovered the missing packages with every make, which took a while to discover.

5.2 Setting Up WiFi

After flashing the Nexus 4, we turned WiFi on and refreshed to see available networks, but the WiFi screen froze. We attempted twice more before assuming we installed the WiFi driver incorrectly and rebuilt mako again.

We attempted to replace the `/persist/wifi/.macaddr` in case the file was missing, but the problem persisted. The logcat revealed that the permissions for `wpa_supplicant.conf` needed to be changed to the same ownerships as `system.wifi`.

```
E/WifiHW  (  550): Cannot set RW to
  "/data/misc/wifi/wpa_supplicant.conf":
  Operation not permitted
E/WifiHW  (  550): Wi-Fi will not be
  enabled

$ adb shell
root@mako# cd /data/misc/wifi
root@mako# chown system.wifi
  wpa_supplicant.conf
root@mako# reboot
```

However, we then encountered the following error in the logcat logs:

```
I/wpa_supplicant(  684): Successfully
  initialized wpa_supplicant
I/wpa_supplicant(  684): rfkill: Cannot
  open RFKILL control device
E/wpa_supplicant(  684): nl80211: Could not
  configure driver to use managed mode
E/wpa_supplicant(  684): Could not read
  interface p2p0 flags: No such device
E/wpa_supplicant(  684): p2p0: Failed to
  initialize driver interface
```

Checking the kernel messages, we find the following message:

```
root@mako:/ # dmesg | grep -i lan
<6>[    1.226827] wcnss_wlan probed in
  built-in mode
<6>[   63.127300] wcnss_wlan triggered by
  userspace
<3>[  124.168228] wcnss_wlan wcnss_wlan.0:
  Peripheral Loader failed on WCNSS.
```

The error occurs in `drivers/net/wireless/wcnss/wcnss_wlan.c`:

```
/* trigger initialization of the WCNSS */
penv->pil = pil_get(WCNSS_PIL_DEVICE);
if (IS_ERR(penv->pil)) {
  dev_err(&pdev->dev, "Peripheral Loader
    failed on WCNSS.\n");
  ret = PTR_ERR(penv->pil);
  penv->pil = NULL;
  goto fail_pil;
}
```

WCNSS is the Qualcomm platform driver used for performing the cold boot-up of the wireless device. We tried building mako again but encountered the same issue.

Several people have had success with downloading a stock ROM in order to fix this issue, but in our case that is not possible because we need to use the ROM we built with TaintDroid [33].

We then looked into sideloading reverse tethering applications, but a majority were incompatible with Mac. We then downloaded HoRNDIS, a USB tethering driver, on our local system [39]. But due to the WiFi driver on the phone, HoRNDIS could not successfully reverse tether the connection.

We downloaded and flashed the factory radio firmware from `developer.google.com`.

```
$ fastboot flash radio radio-mako-m9615a-
  cefwmazm-2.0.1700.84.img
```

Then, the WiFi interface returned no errors in the log and began working smoothly!

5.3 Setting up TaintDroid

Upon flashing the device, we get a notification from TaintDroid Notify. We open the application to view a Start/Stop menu, but the screen does not change when either is pressed. The logs show that the function calls in the TaintDroid Notify application are missing a user handle in the function call:

```
W/ContextImpl( 1497): Calling a method in
  the system process without a qualified
  user: android.app.ContextImpl
  .startService:1385 android.content
  .ContextWrapper.startService:473
```

```
  org.appanalysis
  .TaintDroidNotifyController$1.onClick:16
  android.view.View.performClick:4240
  android.view.View$PerformClick.run:17721

W/ContextImpl( 1497): Calling a method in
  the system process without a qualified
  user: android.app.ContextImpl
  .stopService:1391 android.content
  .ContextWrapper.stopService:478
  org.appanalysis
  .TaintDroidNotifyController$2.onClick:23
  android.view.View.performClick:4240
  android.view.View$PerformClick.run:17721
```

Several people attempting to use version 4.3 of TaintDroid Notify experienced the same issue, which was resolved by downgrading to version 4.1.1 [26].

5.4 Installing Gboard

Since we initially could not connect to WiFi, we downloaded the Gboard apk from apkmirror and installed it via adb [2].

```
$ adb install com.google.android
  .inputmethod.latin_6.0.65.141378828-
  armeabi-v7a-25606516_minAPI17\(armeabi-
  v7a\)\(nodpi\)_apkmirror.com.apk
```

After the enabling Gboard as input in settings, Gboard immediately crashed, followed by Settings.

The error for Gboard in logcat is:

```
W/dalvikvm( 2694): No implementation found
  for native Lcom/google/android/keyboard/
  client/delight4/Decoder;
  .createOrResetDecoderNative:([B)J
D/AndroidRuntime( 2694): Shutting down VM
W/dalvikvm( 2694): threadid=1: thread
  exiting with uncaught exception
  (group=0x415dbe40)
E/AndroidRuntime( 2694): FATAL EXCEPTION:
  main
E/AndroidRuntime( 2694): java.lang
  .UnsatisfiedLinkError: Native method not
  found: com.google.android.keyboard.client
  .delight4.Decoder
  .createOrResetDecoderNative:([B)J
E/AndroidRuntime( 2694): at com.google
  .android.keyboard.client.delight4.Decoder
  .createOrResetDecoderNative(Native
  Method)
```

This error occurs when there is an error within the application; when the libraries are not in the expected directories or the package and class names are mismatched [**?**, 34]. The Gboard apk required at minimum Android 4.2 and its target was Android 7.1. We do not think there is an issue with the apk because other users have successfully installed it.

Once we gained network connection, we downloaded Gboard via the Play Store. It then produced the following errors:

```
D/dalvikvm(20617): Trying to load lib /data/
  app-lib/com.google.android.gms-1/
  libleveldbjni.so 0x42214be0
W/dalvikvm(20617): Denying lib /data/app-
  lib/com.google.android.gms-1/libleveldbjni
  .so (not "/system" prefix)
W/NativeLibraryUtils(20617): Unable to load
  native code from /data/app-lib/com.google
  .android.gms-1/libleveldbjni.so
W/NativeLibraryUtils(20617): java.lang
  .UnsatisfiedLinkError: unknown failure
W/NativeLibraryUtils(20617):  at java.lang
  .Runtime.load(Runtime.java:330)
```

The Java native interface does not allow libraries that are not loaded in from a subdirectory of system as to disallow third party native libraries. We are unable to change this since we do not have the source code for Gboard.

We were unable to find any similar instances of users attempting to install Gboard onto the Nexus 4.

We installed a random wallpaper application to test if it was only Gboard that had linker issues. The wallpaper application also crashed with the same linker error but for a different library.

This is an issue with the Google Play update on Jellybean. The issue has been marked as fix and Google claims it only affects a group of Android phones under specific conditions, but many developers are still experiencing the issue with no workaround [18].

6 Results

The results are that the instructions provided to build Android 4.3 for Nexus 4 (mako) with TaintDroid Notify version 4.3.1 are not sufficient to successfully run TaintDroid Notify. The TaintDroid Notify application needed to be downgraded to version 4.1.1. The issue we encountered was added to the issue tracker in 2014 and marked with medium priority. It was not resolved and the solution we found was in the TaintDroid Google Groups discussion forum.

The TaintDroid project explicitly states that "TaintDroid is a research prototype and is provided "as is" without warranty or support of any kind, whether expressed or implied" [3]. All issues on the issue tracker have not been resolved and questions on the forum have not been answered by TaintDroid's authors since approximately 2013.

Applications cannot run on the TaintDroid Jelly Bean ROM due to the implementation for taint propagation. The lack of support for Android Jelly Bean and its issues prevent applications from being successfully examined.

7 Limitations/Future Work

We were limited to Android 4.2-3 because we were experimenting on the Nexus 4. The Nexus 4 is not backwards compatible with versions of Android below 4.2. We should have experimented with the Galaxy Nexus, the Nexus S, and the Nexus One until we found a ROM that could successfully run TaintDroid Notify. However, the Nexus 4 was the only physical device that we could acquire for this experiment.

We were also limited by the availability of apks to download. Because we were initially unable to connect the device to WiFi, we could not download applications from the Play Story. We relied on old apks that were hosted off the official Play Store. Even after being able to connect to the Play Store, the applications were unable to run due to an unresolved linker error that Google claimed was fixed.

As of 2017, the most recent version of Android is Android 8.1. Android 4.4.4 and below are not supported. A good follow up TaintDroid project would be TaintDroid releases for Android 5-8 as a foremost goal and 4.4.4 as a side goal. In addition, ROMs of TaintDroid on its default setting could be publicly shared so users would only have to install their applications to analyze. Also, TaintDroid Notify notifications could be recorded to a log file for users to extract instead of being notifications within an application.

TaintDroid open to taint evasion through control flow. In order to minimize false positives, TaintDroid chooses to ignore control flow. TaintDroid is able to leverage on-demand static analysis in order to determine control flow graphs. For more development on TaintDroid, control flow analysis should be implemented and disabled by default, allowing users to pursue more in depth analysis if they need to [9].

TaintDroid's main practical issue is that any user will need to build and flash the TaintDroid ROM in order to use TaintDroid Notify. Due to its low level taint tracking, TaintDroid cannot be a stand alone app. TaintDroid stores and tracks taints with variable level tracking within the VM interpreter. If TaintDroid could store and track taints external to the VM and Android system, it would be able to avoid the need to rebuild TaintDroid for every new Android version.

8 Conclusion

Android keyboard applications offer users a customizable experience via multiple features. The rise of keyboard features led to an increase in permissions for application installations. However, the usage of the permissions are opaque. TaintDroid Notify is an application developed in order to provide transparency into the usage of permissions in other applications. We attempted to reproduce TaintDroid's application analysis with one keyboard application, Gboard, in order to test for potential privacy violations. We conducted our experiments on the Nexus 4, Android 4.3 with TaintDroid 4.1.1.

We encountered difficulty building the ROM because the process was not well documented by either Google or the TaintDroid team. We also found that TaintDroid Notify 4.3 had errors with its function calls, so we downgraded to TaintDroid 4.1.1, which functioned as expected. We were unable to install applications due to linker errors from Google Play, which Google claimed to have fixed. The linker errors still persist and there is no fix for those who do not have the source code for the problematic applications.

For future projects with TaintDroid, we recommend following their exact setup. The experiment was tested multiple times on Android 2.1 on multiple Nexus One devices. We also recommend using the ROMs provided by other members of the Google Group, since they have succeeded in successfully using TaintDroid [38].

We will release detailed instructions on how to build the ROM on Github and the TaintDroid Google Group for others to reproduce our steps. We will also contact the authors of TaintDroid for their opinion on continuing TaintDroid for current Android versions.

References

[1] Angwin, Julia, and Jennifer Valentino-DeVries. *Google's iPhone Tracking*. Wall Street Journal. 2012.

[2] APKMirror. *Gboard - the Google Keyboard 6.0.65.141378828*. APKMirror. 2016.

[3] App Analysis Staff. *TaintDroid Build Instructions for Android 4.3*. App Analysis. 2013.

[4] Cai, Liang, and Hao Chen. *TouchLogger: Inferring Keystrokes On Touch Screen From Smartphone Motion*. Proceedings of the 6th USENIX Conference on Hot Topics in Security. 2011.

[5] Cavallaro, Lorenzo, Prateek Saxena, and R. Sekar. *Anti-Taint-Analysis: Practical Evasion Techniques Against Information Flow Based Malware Defense*.

Secure Systems Lab at Stony Brook University, Tech. Rep. 2007.

[6] Cho, Junsung, Geumhwan Cho, and Hyoungshick Kim. *Keyboard or Keylogger?: a security analysis of third-party keyboards on Android*. 13th Annual Conference on Privacy, Security and Trust. 2015.

[7] Clause, James, Wanchun Li, and Alessandro Orso. *Dytan: A Generic Dynamic Taint Analysis Framework*. Proceedings of the 2007 International Symposium on Software Testing and Analysis. 2007.

[8] Dev Academy. *Android Stack*. GitHub Wiki. 2014.

[9] Enck, William, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. *TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones*. Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. 2010.

[10] "Finid". *How to Install and Configure VNC on Ubuntu 16.04*. DigitalOcean. 2016.

[11] Fleishman, Glenn. *Google's Gboard doesn't send your keystrokes, but it does leak chicken and noodles*. Macworld. 2016.

[12] Fritz, Christian, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. *FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps*. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2014.

[13] Fritz, Christian, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. *Highly Precise Taint Analysis for Android Applications*. EC SPRIDE Technical Report. 2013.

[14] Google Inc. *Driver Binaries for Nexus and Pixel Devices*. Google APIs for Android. 2017.

[15] Google Inc. *Federated Learning: Collaborative Machine Learning without Centralized Training Data*. Google Research Blog. 2017.

[16] Google Inc. *The Machine Intelligence Behind Gboard*. Google Research Blog. 2017.

[17] Google Inc. *Search Google & send results from your keyboard*. Google. 2017.

[18] Google Issue Tracker Community. *Loading native libraries fails on 4.1.1 when people update their app*. Google Issue Tracker. 2015.

[19] Griffiths, David. *How Android Apps are Built and Run*.

[20] Hornyack, Peter, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. *"These Aren't the Droids You're Looking For" Retrofitting Android to Protect Data from Imperious Applications*. Microsoft Research. 2011.

[21] Jung, J., A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno. *Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing*. Proceedings of ACM CCS. 2008.

[22] Li, Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick D. McDaniel. *I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis*. CoRR. abs/1404.7431. 2014.

[23] Mohsen, Fadi, and Mohammed Shelab. *Android Keylogging Threat*. 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing. 2013.

[24] Newsome, James, and Dawn Song. *Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software*. Proceedings of the Network and Distributed Systems Symposium. 2005.

[25] "Pulser˙G12" *SwiftKey and Google Keyboard: Ever Heard of User Privacy?* XDA Developers. 2014.

[26] Qi, Wen. *Calling a method in the system process without a qualified user: android.app.ContextImpl.startService*. Google TaintDroid Group Forums. 2015.

[27] Razaghpanah, Abbas, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Christian Kreibich, Phillipa Gill, Mark Allman, and Vern Paxson. *Haystack: A Multi-Purpose Mobile Vantage Point in User Space*. N.p. 2015.

[28] Sarwar, Golam, Olivier Mehani, Roksana Boreli, and Mohamed-Ali Kaafar. *On the Effectiveness of Dynamic Taint Analysis for Protecting Against Private Information Leaks on Android-based Devices*. 2013 International Conference on Security and Cryptography. 2013.

[29] Sasnauskas, Raimondas, and Regehr, John. *Intent Fuzzer: Crafting Intents of Death*. Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA), WODA+PERTEA 2014. 2014.

[30] Schüette, J., A. Küechler, and D. TItze. *Practical Application-Level Dynamic Taint Analysis of Android Apps*. 2017 IEEE Trustcom/BigDataSE/ICESS. 2017.

[31] Schwartz, Edward J., Thanassis Avgerinos, and David Brumley. *All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)*. Proceedings of the IEEE Symposium on Security and Privacy. 2010.

[32] Shankar, Venkatesh Gauri, Gaurav Somani, Manoj Gaur, Vijay Laxmi, and Mauro Conti. *AndroTaint: An Efficient Android Malware Detection Framework Using Dynamic Taint Analysis*. Proceedings of the IEEE Symposium on Asia Security and Privacy. 2017.

[33] Stack Exchange Community. *Nexus 4: Stuck on boot animation for baseband version != 2.0.1700.48 and android version ¿ 4.3*. Stack Exchange. 2015.

[34] Stack Overflow Community. *Android NDK java.lang.UnsatisfiedLinkError: findLibrary returned null*. Stack Overflow. 2016.

[35] Stack Overflow Community. *JNI-java.lang.UnsatisfiedLinkError: Native method not found*. Stack Overflow. 2017.

[36] Wagoner, Ana, and Marc Lagace. *Best Keyboard for Android*. Android Central. 2017.

[37] Welton, Ryan. *Remote Code Execution as System User on Samsung Phones*. Now Secure. 2015.

[38] Wen, Yan. *CM-10 ROMs (with TaintDroid integrated) for 22 mobile phones*. Google TaintDroid Group Forums. 2013.

[39] Wise, Joshua. *HoRNDIS: USB tethering driver for Mac OS X*. Joshua Wise. N.d.

[40] You, Wei, Bin Liang, Wenchang Shi, Peng Wang, Xiangyu Zhang. *TaintMan: an ART-Compatible Dynamic Taint Analysis Framework on Unmodified and Non-Rooted Android Devices*. IEEE Transactions on Dependable and Secure Computing. 2017.

[41] Yumerefendi, A.R., B. Mickle, and L.P. Cox. *TightLip: Keeping Applications from Spilling the Beans*. Proceedings of the 4th USENIX Symposium on Network Systems Design & Implementation. 2007.

[42] Zeltser, Lenny. *Security of Third-Party Keyboard Apps on Mobile Devices*. Lenny Zeltser. 2017.

[43] Zorz, Mirko. *Gboard enhances your keyboard, but what about your privacy?* Help Net Security. 2016.