PrivacyGuard: A VPN-Based Approach to Detect Privacy Leakages on Android Devices

by

Yihang Song

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2015

© Yihang Song 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The Internet is now the most important and efficient way to gain information, and mobile devices are the easiest way to access the Internet. Furthermore, wearable devices, which can be considered to be the next generation of mobile devices, are becoming popular. The more people rely on mobile devices, the more private information about these people can be gathered from their devices. If a device is lost or compromised, much private information is revealed. Although today's smartphone operating systems are trying to provide a secure environment, they still fail to provide users with adequate control over and visibility into how third-party applications use their private data. The privacy leakage problem on mobile devices is still severe. For example, according a field study [1] done by CMU recently, Android applications track users' location every three minutes in average.

After the PRISM program, a surveillance program done by NSA, is exposed, people are becoming increasingly aware of the mobile privacy leakages. However, there are few tools available to average users for privacy preserving. Most tools developed by recent work have some problems (details can be found in chapter 2). To address these problems, we present PrivacyGuard, an efficient way to simultaneously detect leakage of multiple types of sensitive data, such as a phone's IMEI number or location data. PrivacyGuard provides real-time protection. It is possible to modify the leaked information and replace it with crafted data to achieve protection. PrivacyGuard is configurable, extensible and useful for other research.

We implement PrivacyGuard on the Android platform by taking advantage of the `VPNService` class provided by the Android SDK. PrivacyGuard does not require root permissions to run on a device and does not require any knowledge about VPN technology from users either. The VPN server runs on the device locally. No external servers are required. According to our experiments, PrivacyGuard can effectively detect privacy leakages of most applications and advertisement libraries with almost no overhead on power consumption and reasonable overhead on network speed.

Acknowledgements

I would like to thank all people who made this possible.

Dedication

This is dedicated to people I love.

Table of Contents

List of Tables ix

List of Figures x

1 Introduction 1

1.1 Overview 1

1.2 Goals 3

1.3 Contributions 3

2 Related Work 5

2.1 Taint Analysis 5

2.1.1 Dynamic Taint Analysis 6

2.1.2 Static Taint Analysis 8

2.2 Android Retrofitting 9

2.3 App Rewriting 11

2.4 Heuristics 13

2.5 VPN-based Approaches 13

2.6 Summary 13

3 Design and Implementation 16

3.1 Design Options 16

3.2 Different Solutions based on VPN 17
3.2.1 Remote VPN server 17
3.2.2 Local VPN server 17
3.2.3 Fake VPN server 18
3.2.4 Main Difficulties 18
3.3 Framework 18
3.4 Basic Work Flow 19
3.5 FakeVPNService 20
3.5.1 Android VPN Service 20
3.5.2 `FakeVPNService` 20
3.6 `TCPForwarder` 22
3.6.1 TCP Connection States 22
3.6.2 Connection with `LocalServer` 26
3.7 `LocalServer` 28
3.7.1 Connection Accepted 28
3.7.2 How to Determine the Real Server 28
3.8 `SocketForwarder` 28
3.8.1 Handling SSL Connections 29
3.9 Customized Plugins 31
3.9.1 Implemented Plugins 31
3.10 Limitations 32
4 Evaluation **33**
4.1 Performance Evaluation 33
4.1.1 Setup 33
4.1.2 Network Performance 34
4.1.3 Power Consumption 37
4.2 Effectiveness Evaluation 38

4.2.1 Setup 38
4.2.2 Methodology 39
4.2.3 Training 40
4.2.4 Testing Results 41
4.3 Conclusion 43

5 Future Work **44**

6 Conclusion **46**

APPENDICES **47**

A Experiment Data **48**
A.1 Network Performance Experiments 48

B Script for Setting up Google Play Service on Emulators **51**

C Interface for Plugins **52**

D `/proc/net/tcp` Files Example **53**

References **56**

List of Tables

4.1 Decrease in Battery Level for Power Consumption Experiments 38
4.2 Training Apps List 40
4.3 Filters 41
4.4 Testing Results of Applications 42
4.5 Testing Results of Advertising Libraries 42
4.6 Testing Results Analysis 43

A.1 Experiment Data for 1Mbytes file experiments 49
A.2 Experiment Data for 10Mbytes file experiments 49
A.3 Experiment Data for SpeedTest.net 50

List of Figures

1.1 Access Control of Different Platforms 2

2.1 TaintDroid Architecture [8] 7
2.2 Malicious Callback Examples [6] 14
2.3 Architecture of Edgeminer [6] 15
2.4 Decision diagram of LP-Guardian [9] 15

3.1 Architecture of PrivacyGuard 19
3.2 Caption for LOF 22
3.3 Caption for LOF 23
3.4 TCP state machine 25
3.5 Configuration of the Sockets Used in PrivacyGuard 27
3.6 Caption for LOF 30

4.1 Results of Network Performance Evaluations 35

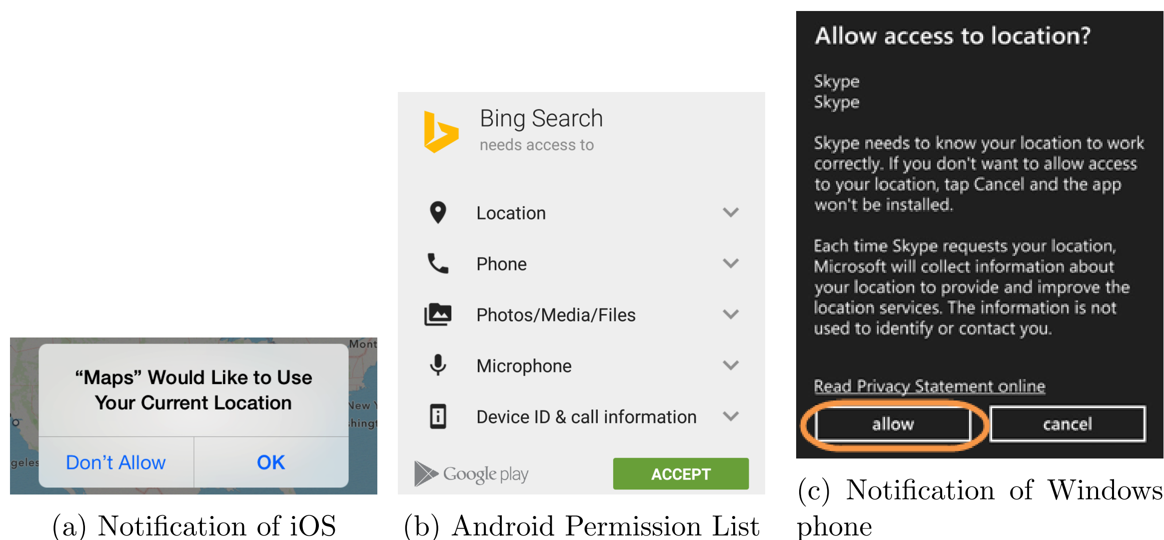
Chapter 1

Introduction

1.1 Overview

Mobile devices such as smartphones and tablets have become popular and powerful. Such devices can have many sensors, for example, gyroscopes, GPS, fingerprint sensors and even heart rate sensors embedded in some wearable devices. These sensors collect a great deal of personal information. Also, since people carry the devices all day and use the devices to communicate or work, these devices can contain much private information. This information makes it possible to track, identify or profile users. All operating systems are trying to build a mechanism to provide secure access to this information for applications, such that this information will not be revealed to malicious people. The App Store has a review process to avoid malicious applications. iOS has a first-time use notification for access to location, microphones among others, as shown in figure 1.1a. Users can also manually set the access policy for each application later in the settings. Android uses a permission-based security model to restrict applications from accessing private data and privileged resources. When users install an application, there will be prompts such as figure 1.1b. Users can grant the permissions or cancel the installation. The Windows Store also has an approval process. When an application wants to access some sensitive data for the first time, there will also be a notification as shown in figure 1.1c. There is also a way to configure the location access of applications in the settings.

Although existing platforms have provided the approaches mentioned above to restrict access and alert users about accesses, these approaches are not efficient, especially when users do not understand what happens. Significant efforts have been made to explore these challenges ([3], [4], [15]).



(a) Notification of iOS (b) Android Permission List (c) Notification of Windows phone

Figure 1.1: Access Control of Different Platforms

The other problem with the approaches employed by existing platforms is that all of these are all-or-nothing. Users can control only whether an application can access their private data. Intuitively, there would only be privacy leakages if applications send data out. However, these existing mechanisms block access no matter how this data is used or processed before being sent out. These mechanisms may block even if this data is not sent out. For example, an application may access the location to obtain the current time zone, or it may implement some anonymity algorithms (such as [11], [20]), and send out a well-crafted location that does not leak any essential information. These all-or-nothing approaches would then affect the usability of benign applications.

Much existing work cannot address the problems described above. For example, some tools (e.g., [8], [2]) leverage taint analysis, which sets a taint flag on the data returned by privacy-sensitive methods (e.g., `getLastKnownLocation()`) and checks if the data reaching some sinks (e.g., `Http.request()`) has the flag. These tools cannot handle the problems properly because the taint will be propagated no matter how the data is processed in the flow. Application rewriting approaches (e.g., [7], [12]), which usually modify the application to achieve better access control, however, still do not consider how applications use the private data. These methods usually have some other problems, such as the requirement of root permission, modification of the Android kernel, or problems when updating applications. The details are discussed in chapter 2.

1.2 Goals

To solve these problems mentioned above, we present a privacy-preserving application that alerts the user when it detects leakages of sensitive information. We present the goals of our design below:

- **Functionality:** This application should be able to detect leakages of private information. When it detects any leakage, it notifies the user of which application is leaking what information. It should also support the analysis of SSL traffic.
- **Usability:** This application should not require root permissions and should work like a typical application. It should not require much configuration or any specific knowledge about security and privacy from users. It should be portable to any Android version greater than Android 4.0 and any Android device.
- **Acceptable Overhead:** This application runs continuously on the device. Therefore, it should not use too much battery power. Since it provides real-time protection by analyzing the network traffic, it should not affect the network performance much.
- **Extensibility:** This application should be extensible. Developers can develop their plugins and take advantage of the access to network traffic provided by this application. Also, researchers can use this application to prototype their designs or algorithms.

1.3 Contributions

We present PrivacyGuard, a privacy-preserving application that tries to provide a way to help users control their private data. As far as we know, PrivacyGuard is the first open-source Android application that uses Android's `VPNService` without any remote VPN server to do network traffic sniffing. It analyzes network traffic in real time and checks if it contains any private data. Since Android is one of the most popular mobile operating systems, we implement PrivacyGuard as an Android application. However, it is possible to port PrivacyGuard to other platforms.

In summary, we make the following contributions:

- We present the design of PrivacyGuard, which we believe to be the first privacy-preserving application that does customized filtering on network traffic.

- We implement PrivacyGuard based on our design.
- PrivacyGuard provides a framework for other research or development to obtain access to network traffic.
- PrivacyGuard introduces little overhead in power consumption and acceptable overhead in network performance.
- PrivacyGuard can detect privacy leakages of most applications effectively and has better results than TaintDroid [8], a popular taint-based approach.

During the development of PrivacyGuard, we found a proxy-based network interception tool called SandroProxy[1]. We had several discussions with its author and benefitted from his experience in developing SandroProxy. The author of SandroProxy also planned to develop a VPN-based network interception. The source code of PrivacyGuard is shared with him, and he provided advice about fixing problems in PrivacyGuard.

In this thesis, we present the design and implementation of PrivacyGuard. Chapter 2 introduces some related work. Chapter 3 explains the design and implementation. Chapter 4 shows the overhead introduced by PrivacyGuard and that it can effectively detect privacy leakages of most applications. Possible future work is discussed in chapter 5, and we conclude in chapter 6.

[1] https://code.google.com/p/sandrop/

Chapter 2

Related Work

This section presents the related work in this area. The related research to our work can be classified into several categories by approach. Section 2.1 focuses on prior work that takes advantage of taint analysis to detect privacy leakages. Section 2.2 presents some work that modifies the Android operating system to preserve privacy. Section 2.3 describes some work that tries to analyze and modify the code of existing applications to generate a more secure version. In section 2.4, some research that uses heuristic approaches is discussed. In section 2.5, some work that also uses the `VPNService` is discussed.

2.1 Taint Analysis

Taint analysis detects privacy leakages by analyzing applications. There are some fundamental concepts used in taint analysis:

- **Source method:** pre-defined methods (defined by developers) that can be called to retrieve sensitive data, e.g., `getLastKnownLocation()`
- **Sink method:** pre-defined methods (defined by developers) that can potentially leak sensitive data, e.g., `Http.request()`
- **Taint:** a flag associated with a variable. This flag means the data stored in the variable is sensitive.
- **Propagation:** rules of taint passing between variables

Developers must define all the concepts above before executing taint analysis. The analysis tool adds taint to the data returned by any source methods and tracks how the data flows until the data reaches a sink method or goes to some end. In general, there are two kinds of approaches to do taint analysis: dynamic taint analysis and static taint analysis.

2.1.1 Dynamic Taint Analysis

Dynamic taint analysis monitors the data flow simultaneously while applications are running. The real-time monitoring is often done by modifying the Android kernel. The code of all source methods is changed to add taint flags to the return values. In all other methods or operators whose output should have taint flags if their input has taint flags, code for taint propagations is added. Sink methods are also modified to check if the arguments, class members or static variables that these methods use are tainted. Dynamic taint analysis works well, but it has several problems that prevent it from being used by average users.

- **Advantages:**
  - Only reports when there exists a path from a source to a sink while the application is running
  - Runs online, is easy to use once set up
- **Disadvantages:**
  - Will report leakages even if the processed data is no longer sensitive
  - Is usually implemented as a ROM. It needs to flash the device and is difficult to use.
  - Usually requires root permissions and may introduce other vulnerabilities
  - Is difficult to port to different devices or different Android versions
  - Can be detected by some current malicious applications
  - Is difficult to add new source methods or sink methods
  - Requires multiple runs to reach appropriate code coverage

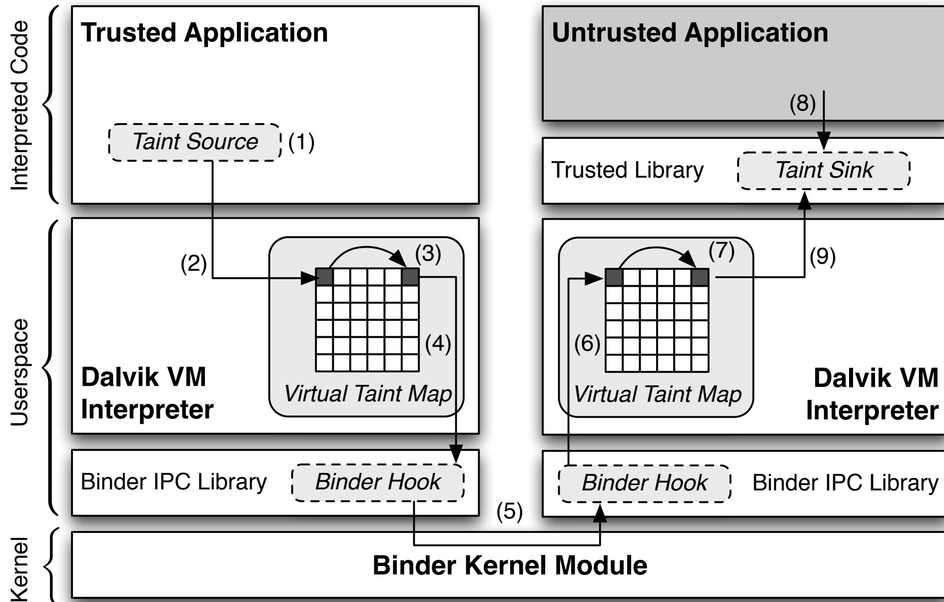


Figure 2.1: TaintDroid Architecture [8]

TaintDroid

TaintDroid [8] is a modified Android kernel. Its main architecture is shown in figure 2.1. TaintDroid sets taint flags on objects returned by source methods and also implements taint propagation rules. When a tainted object is used as a parameter of a sink method, TaintDroid will report a privacy leak.

TaintDroid has some limitations; it cannot handle native libraries, and it requires root permissions. There have been some easy approaches that can bypass the check of TaintDroid, such as AntiTaintDroid[1]. When we used TaintDroid, we found there are many superfluous reports that we think are false positives.

BayesDroid

BayesDroid [18] is also a modified Android kernel. It leverages the dynamic taint analysis approaches. The authors identified that there is a high false positive rate in traditional information flow analysis because existing tools do not take the possible masking on sensitive

[1] http://gsbabil.github.io/AntiTaintDroid/

data into consideration. For example, even if only the country information retrieved from location data is sent out, old approaches will still report a location leakage because there is a data-flow of tainted data from a source to a sink. BayesDroid does Bayes classifications on data reaching sink methods. It computes the distance (e.g., Hamming Distance) between the data to be sent and the original sensitive data. Given the distance, BayesDroid computes the probability that the sharing is illegitimate or legitimate.

2.1.2 Static Taint Analysis

Static taint analysis usually decompiles application APKs first to obtain the source code of these applications. Based on the code, static taint analysis tools reconstruct control flow graphs based on the source code and build a model of the run-time execution. With the model, the tools can detect if there are any data flows from source methods to sink methods statically.

- **Advantages:**
  - Static analysis can have a deeper analysis of the code in one run
- **Disadvantages:**
  - Runs offline. Is difficult to be used by average users.
  - Takes too much time to obtain accurate results
  - Has a high false positive rate since many detected flows will not happen in real use
  - Can be imprecise because it needs to abstract from program inputs and to approximate run-time objects

FlowDroid

FlowDroid [2] does static analysis on the source code of applications. It first builds a call graph. Based on the graph, FlowDroid does backward data analysis to find where the data reaching a sink method comes from. If the data comes from a defined source method, FlowDroid will mark it as a leakage. One advantage of FlowDroid is that it also takes the life cycle of Android components into consideration, e.g., `Activity.onCreate()`. FlowDroid handles these components by analyzing the `AndroidManifest.xml` file, extracting

all registered Android components and building a corresponding call graph based on the documentation of Android.

FlowDroid has several limitations; it takes too much time for a deep analysis, which is a common problem for static analysis tools. In addition, it does not take the callback methods, e.g., `onLocationChange()`, into consideration.

EdgeMiner

EdgeMiner [6] is an enhancement recently proposed. It can be integrated into existing static analysis tools. The authors identify that one important reason for the imprecision of existing static analysis tools is the inadequate consideration of callback (e.g., `onLocationChange()`) and register (e.g., `setOnClickListener()`) methods. Figure 2.2 gives two examples of how callback methods leak private information. The call graph built by general static analysis tools will not contain callback methods because these methods are called by the Android framework, not the application. If that method contains some unexpected malicious code, EdgeMiner can still detect it. Although the Android framework declares many callback methods, there is no available documentation for these methods. One of the most important contributions of EdgeMiner is that it can extract callback and register methods for the Android framework.

The architecture of EdgeMiner is shown in figure 2.3. EdgeMiner uses some heuristic rules, such as the function signature, to search for potential callback and register methods. For each potential callback method, EdgeMiner uses backward data-flow analysis to find if there is a way back from the method to any potential register method. With the callback and register method information, static taint analysis tools can build a more accurate model of the run-time execution.

2.2 Android Retrofitting

Although researchers usually modify Android source code as well in dynamic taint analysis tools, in this section, we focus on other approaches that use OS-based techniques to achieve their goals. Existing work usually adds the support for intercepting the location access and modifies the location data passed to applications.

- **Advantages:**

  - No need to modify existing applications. In theory, the operating system should support all existing applications.

- **Disadvantages:**
  - Needs to flash the device. Is difficult for average users. May be unstable.
  - Requires rooting the device
  - May not be portable to new Android versions. Even if portable, it may require flashing again for upgrading. Inability to upgrade is a significant problem because Android updates usually fix many security issues.
  - Is inflexible and difficult to update the security policies. Is difficult to customize security policies for different applications.

AppFence

With AppFence [14], users can withhold data from imperious applications that demand information that is unnecessary to perform their functionalities. AppFence takes advantage of TaintDroid [8] and provides two privacy controls: shadow data and exfiltration blocking. When an application demands access to sensitive data a user does not want the application to have, AppFence will substitute an innocuous shadow in its place. For example, AppFence may replace IMEI with the hash of IMEI. Shadow data may break applications that truly require this data to provide functionality, e.g., a contact app has to access the contact data. For the data that is allowed to be accessed, AppFence will prevent it from being sent out.

LP-Guardian

LP-Guardian [9] provides privacy protection on a per-app basis. The protection works according to the running states of the application. The principles are shown in figure 2.4. LP-Guardian analyzes the calling stack trace to determine whether a location request is from the core app or the advertising library the application includes. To manipulate the location data, LP-Guardian modifies the platform instead of application rewriting. It adds application context data into the `Location` class and adds a location interceptor that will be informed whenever there is a location update. Based on the security policy, LP-Guardian executes different operations in different situations.

2.3 App Rewriting

Application rewriting makes modifications to applications to achieve the desired security policies. It is usually done via Java bytecode rewriting or Dalvik bytecode rewriting.

- **Advantages:**
  - Leaves the underlying platform unmodified. Does not require flashing or rooting the device.
  - Provides flexible security policies. It is possible to set different policies for different applications.
  - Supports different Android versions if the corresponding APIs are not modified
- **Disadvantages:**
  - Is not as powerful as OS-based modifications
  - Users have to trust other people's modifications or know how to set up their own policies.
  - Is inconvenient for updating applications since rewriting will break the signature of the application. Also, users need to do the rewriting again after updating.

I-ARM

Davis et al. [7] propose a rewriting framework to embed In-App Reference Monitors (I-ARM) in Android applications. **A reference monitor** concept defines a set of design requirements on a reference validation mechanism, which enforces an access control policy over subject ability to perform operations on objects in a system. With I-ARM, users can identify which methods of the Android framework that they want to change by specifying the full method signature. Users can add custom behaviors to each of these methods. I-ARM also supports rewriting methods called by reflection but not called by native libraries. This approach provides flexible method-level rewriting. However, it is tedious to rewrite every security-related method. Furthermore, it does not take into consideration if the data is sent out.

Dr. Android and Mr. Hide

Jeon et al. [15] identify that the existing Android permission model is coarse-grained. Available permissions are often much more powerful than necessary. They propose a new fine-grained permission model and a tool chain: **RefineDroid**, **Mr. Hide**, and **Dr. Android**. RefineDroid does statistic analysis on Android applications to infer fine-grained permissions based on their model. To enforce the inferred permissions without modifying the Android platform or rooting the device, they introduce Mr. Hide, a set of Android services that wrap several privileged Android APIs. These services run in their own thread and expose some new APIs that require fine-grained permissions to be called. With Mr. Hide, developers can easily use fine-grained permissions. For existing applications, Dr. Android can retrofit these applications to enforce these fine-grained permissions by modifying the APK and changing method calls to the new APIs in Mr. Hide.

AppGuard

Backes et al. [5] use caller-site rewriting to inline the reference monitor into existing third-party apps. For caller-site rewriting, it means application binaries are rewritten to make sure a security monitor is invoked at run-time before a security-related function call. The security monitor dynamically checks if all enforced security policies allow the attempted operations. AppGuard is implemented as a stand-alone application. Average users can use AppGuard easily and do not need to do the rewriting on a desktop manually. AppGuard supports both reflective calls and calls from native libraries to Java methods. However, it is not able to monitor function calls inside native libraries.

Idea

Idea [19] uses callee-site rewriting to inline the reference monitor into existing applications. Unlike caller-site rewriting, which does the security checks before the security-relevant methods are called, callee-site rewriting only instruments the entry of these methods. Since Android libraries are sealed, and doing direct callee-site rewriting is infeasible, Idea uses a new technique called *call diversion*. Call diversion reroutes the function calls from the security-relevant APIs to methods defined in Idea. These methods invoke the security monitor to enforce access control policies.

2.4 Heuristics

Fu et al. [10] use a heuristic way to provide run-time location access disclosures. According to the documentation of `getLastKnownLocation()`, they devise two heuristics. (1) The return value of `getLastKnownLocation()` will change if and only if any application is requesting location updates; (2) the most likely application requesting the location is the app the user is actively using. However, these heuristics are not entirely correct since applications may use `getLastKnownLocation()` themselves to obtain location fixes. Calling `getLastKnownLocation()` will not be noticed by this heuristic approach because the return value of that function does not change after calling. Also, applications registering a location listener can obtain location fixes in the background.

2.5 VPN-based Approaches

Disconnect

Disconnect[2] uses the `VPNService` class. The Disconnect application blocks all communications between all other applications running on the device and advertising servers. The method the Disconnect application uses is setting up a DNS server that responds with `127.0.0.1` to all queries for hosts of advertising servers.

2.6 Summary

All of this related work has advantages. However, there are some obvious disadvantages compared to PrivacyGuard. (a) Most of this work does not consider whether the accessed data is sent out or in what form the data is sent out. The lack of consideration will cause false positives and affect the usability; (b) some of this work requires rooting or flashing the device. Rooting is usually considered to be a significant source of insecurity since applications with root permissions can break the device easily. Flashing the device is both difficult and inconvenient for average users; (c) furthermore, application rewriting will cause problems when updating applications.

PrivacyGuard does not have these problems. It works as a typical Android application. Updating PrivacyGuard is easy and does not require root permissions. PrivacyGuard runs continuously on the device and provides real-time protections.

[2] https://disconnect.me/

```
1  class MainClass {
2    static int value = 0;
3    static void main(String[] args) {
4      MalComp mal = new MalComp();
5      MainClass.value = 42;
6      Collections.sort(list, mal);
7      sendToInternet(MainClass.value);
8    }
9  }
10 class MalComp implements Comparator {
11   int compare(
12         Object arg0,
13         Object arg1) {
14     MainClass.value = getGPSCoords();
15     return 0;
16   }
17 }
```

(a) Application Space

```
1  public interface Comparator<T> {
2      public int compare(T lhs, T rhs);
3  }
4
5  public class Collections {
6    public static <T> void sort(
7          List<T> list,
8          Comparator<? super T> comparator) {
9      ...
10     comparator.compare(element1, element2);
11     ...
12   }
13 }
```

(b) Framework Space

(a) Synchronous callback methods. The **MalComp.compare()** method will be called by **Collections.sort()**, which is defined in the Android framework. However, there is no explicit calling of this method in application source code and thus difficult to be noticed by previous static analysis tools.

```
1  class MainActivity extends Activity {
2    static int value = 0;
3    onCreate(Bundle bundle) {
4      MalListener mal = new MalListener();
5      MainActivity.value = 42;
6      // get a reference to a button GUI widget
7      Button b = [...]
8      b.setOnClickListener(mal);
9    }
10 }
11 class FinalActivity extends Activity {
12   // This activity is reached towards the
13   // end of the application's execution.
14   onCreate(Bundle bundle) {
15     sendToInternet(MainActivity.value);
16   }
17 }
18 class MalListener implements OnClickListener {
19   int onClick(View v) {
20     MainActivity.value = getGPSCoords();
21     return 0;
22   }
23 }
```

(a) Application Space

```
1  public class ViewRootImpl extends Handler {
2    public void handleMessage (Message msg) {
3        switch (msg.what) {
4          case EVENT:
5            mView.mOnClickListener.onClick();
6          ...
7        }
8    }
9  }
10 public class View {
11   OnClickListener mOnClickListener;
12   public void setOnClickListener (EventListener
         li) {
13       mEventListener = li;
14   }
15 interface OnClickListener {
16   void onClick(View v) {
17   }
18 }
```

(b) Framework Space

(b) Asynchronous callback methods. The **MalListener.onClick()** is called by **ViewRootImpl.handleMessage()**, which is executed in the UI thread. Apart from no explicit calling of the method in the application, the calling is actually asynchronous and difficult to reveal.

Figure 2.2: Malicious Callback Examples [6]

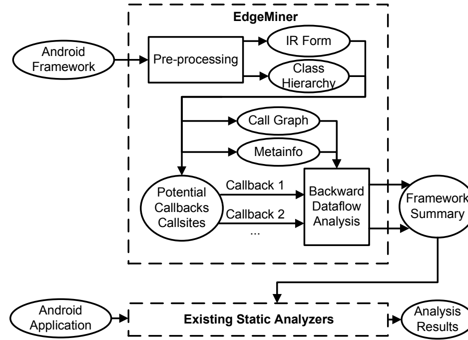


Figure 2.3: Architecture of Edgeminer [6]

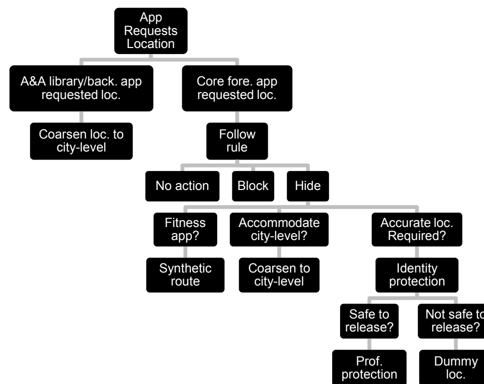


Figure 2.4: Decision diagram of LP-Guardian [9]

Chapter 3

Design and Implementation

3.1 Design Options

An approach to gain access to all network traffic is required, similar to *tcpdump* or *wireshark*. With this access, we can do analysis on the traffic to detect privacy leakages. Since Android is based on Linux, porting *tcpdump* is feasible (in fact, there is a *tcpdump* for Android). However, porting *tcpdump* requires root permissions. The requirement of root permissions is unacceptable to us.

There are two possible ways to gain access. One is by using a proxy, and the other is by using a VPN.

A proxy is a server (a computer system or an application) that acts as an intermediary for requests from clients seeking resources from other servers. Proxies have several types, such as a Web proxy or a SOCKS proxy. A web proxy cannot fulfill our requirements because it can only forward HTTP packets. A SOCKS proxy supports both TCP and UDP. Due to the restrictions of Android, no application can configure the proxy automatically without root permissions. It is necessary to manually configure the proxy for every WiFi connection. The manual configuration can be a problem for average users.

We choose to use a VPN, which refers to virtual private network. VPNs extend a private network across a public network such as the Internet. VPNs are created by establishing a virtual point-to-point connection through the use of dedicated connections. VPN works on the IP layer. An application can programmatically set up a VPN connection that can be used by other applications, and the only user interaction required is tapping OK on the connection prompt once.

3.2 Different Solutions based on VPN

There are several possible solutions based on using a VPN.

3.2.1 Remote VPN server

The easiest way is by setting up a remote VPN server and letting the device connect to it. All network traffic will go through the server. The server can do the analysis and report all privacy leakages to users. A remote VPN server is easy to set up and use, but it has some problems:

1. The network traffic is sent to a remote server, which users may not trust
2. All data is revealed if the server is compromised
3. There may be a high load for the server if there are many users
4. Network overhead is introduced

To avoid these problems, we propose the second option.

3.2.2 Local VPN server

A VPN server that can run locally does not have the problems mentioned above. All network traffic is available only to the device. The approach is promising, but it has the following problem:

- Since the VPN server runs on the IP layer, we need to use raw sockets to transmit IP datagrams directly. However, without root permissions, using raw sockets is not allowed by Android.

Again, rooting the device is unacceptable and we do not need a fully-featured VPN server. We need access to all network traffic, but do not need any other features of a VPN, such as encryption.

3.2.3 Fake VPN server

We can use the `VPNService` class provided by the Android SDK after version 4.0.

The `VPNService` class is a base class for applications to extend and build their own VPN solutions. In general, it creates a virtual network interface, configures addresses and routing rules, and returns a file descriptor to the application. Each read from the descriptor retrieves an outgoing packet that was routed to the interface. Each write to the descriptor injects an incoming packet such as that received from the interface. The interface is running on the IP layer, so packets are always started with IP headers. The application then completes a VPN connection by processing and exchanging packets with the remote server over a tunnel.

In our case, there is no real VPN server. The `VPNService` pretends that we have connected to a VPN server. PrivacyGuard obtains network traffic from the virtual network interface provided by `VPNService` and retransmits it after the analysis. The details can be found in later sections.

3.2.4 Main Difficulties

While developing PrivacyGuard, we need to overcome the following difficulties:

- Because a VPN server works on the IP layer, we need to implement an IP and TCP/UDP stack in Java. The implementation should follow the protocol specifications and also be efficient.
- All analysis requires network traffic in plain text. PrivacyGuard should be able to decrypt packets encrypted with the SSL protocol.

3.3 Framework

As shown in figure 3.1, PrivacyGuard has five main components.

- `FakeVPNService`
- `TCPForwarder` (or `UDPForwarder`)
- `LocalServer`

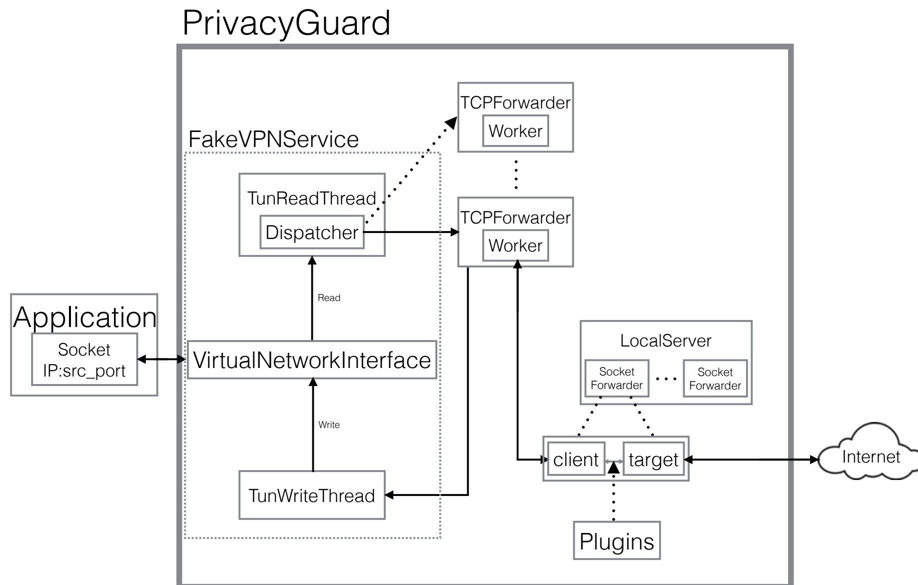


Figure 3.1: Architecture of PrivacyGuard

- `SocketForwarder`
- `Plugins`

Because all components except the forwarders are similar between TCP and UDP and the forwarding for TCP is more important and complex, all explanations are based on TCP unless explicitly specified.

3.4 Basic Work Flow

The following is the basic work flow of PrivacyGuard:

1. An application sends a request to a server. The request can then be retrieved from the virtual network interface in the `FakeVPNService`.

2. The FakeVPNService parses the request contained in an IP datagram and dispatches the request to the corresponding forwarder. In the TCP case, a TCPForwarder will handle the request.

3. The TCPForwarder implements TCP. The TCPForwarder communicates with the LocalServer, which is running on the TCP layer. The LocalServer acts like a man-in-the-middle proxy.

4. For each request received from a TCPForwarder, the LocalServer retransmits the request to the real server and also retransmits the response from the real server to the TCPForwarder. In this step, **all Plugins are invoked to filter both the request and the response to preserve privacy**.

5. The TCPForwarder packages the response from the LocalServer into an IP datagram and sends the datagram through the virtual interface to the application.

All components are explained in detail in the following sections.

3.5 FakeVPNService

3.5.1 Android VPN Service

The VPNService[1] class is added to the Android SDK after API level 14 (Android 4.0). It creates a virtual network interface, configures addresses and routing rules, and returns a file descriptor to the application. From the descriptor, the application can read all network traffic.

3.5.2 FakeVPNService

The FakeVPNService extends the VPNService. The FakeVPNService establishes a virtual network interface with the IP address 10.8.0.1. It also configures routing rules to route network traffic sent to all IP addresses to the interface. After properly setting up the FakeVPNService, the Android system will route all network traffic from all other applications to this virtual network interface in the FakeVPNService. The FakeVPNService sets up several threads. These threads are described below:

[1] http://developer.android.com/reference/android/net/VpnService.html

`TunReadThread`

This thread reads all requests from the virtual network interface. Because there are many applications and services running on the device, the amount of requests can be large. There will be heavy performance overhead due to the high network IO cost if the `TunReadThread` handles the transmission to other components as well. To avoid the cost, this thread only adds these requests to a queue.

`Dispatcher`

This thread is created by the `TunReadThread`. The `Dispatcher` keeps reading requests, which are IP datagrams since VPN works on the IP layer, from the queue mentioned above. The `Dispatcher` parses these IP datagrams to obtain necessary information, such as the IP address.

Each request received from the interface is an IP datagram. The `Dispatcher` retrieves the protocol field from the IP header to see whether the datagram wraps a TCP packet or a UDP packet. If the datagram contains a TCP packet, a `TCPForwarder` bound to the source port number of the packet is used to handle this packet. If there has already been a forwarder for the port number, this forwarder is used. Otherwise, a new forwarder is created and bound to that port number. The reason for using the old forwarder is that a TCP connection is stateful. The forwarder maintains the state of the TCP connection.

`TunWriteThread`

The `TunWriteThread` retrieves responses from a queue and writes these responses to the virtual network interface. The responses are added to the queue by forwarders by calling the `TunWriteThread.write()` method. Because this method requires IP datagrams as arguments, forwarders need to reconstruct valid IP datagrams from TCP/UDP data these forwarders have. The main part of the reconstruction is reversing the source and destination IP addresses in the request and updating the IP checksum.

Concurrent access to two queues

The two queues in the `TunReadThread` and `TunWriteThread` are accessed and modified concurrently by more than two threads. This concurrent access is a typical producer-consumer problem. Proper synchronization is required to avoid concurrency issues. Initially, we

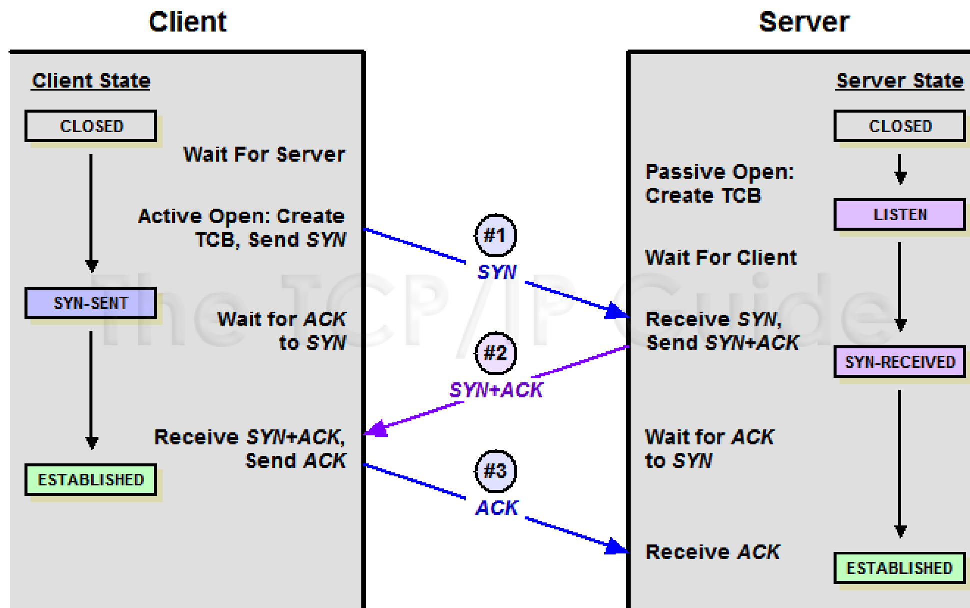


Figure 3.2: TCP Connection Establishment[2]

implemented these two queues with `ArrayLists` and did synchronization manually by using `synchronize`, `wait()`, and `notify()`. However, during the performance evaluation, we found that this solution would slow down the transmission. Java has a class called `ConcurrentLinkedQueue`. It provides efficient thread-safe access. There is a significant improvement in the performance after switching to this class.

3.6 TCPForwarder

3.6.1 TCP Connection States

TCP is designed to provide reliable, ordered and error-checked delivery of a stream of octets between programs. It works on the transport layer and provides an end-to-end connection.

A TCP connection mainly has three states:

1. Connection establishment (see figure 3.2)

[2] http://www.tcpipguide.com/free/t_TCPConnectionEstablishmentProcessTheThreeWayHandsh-3.htm

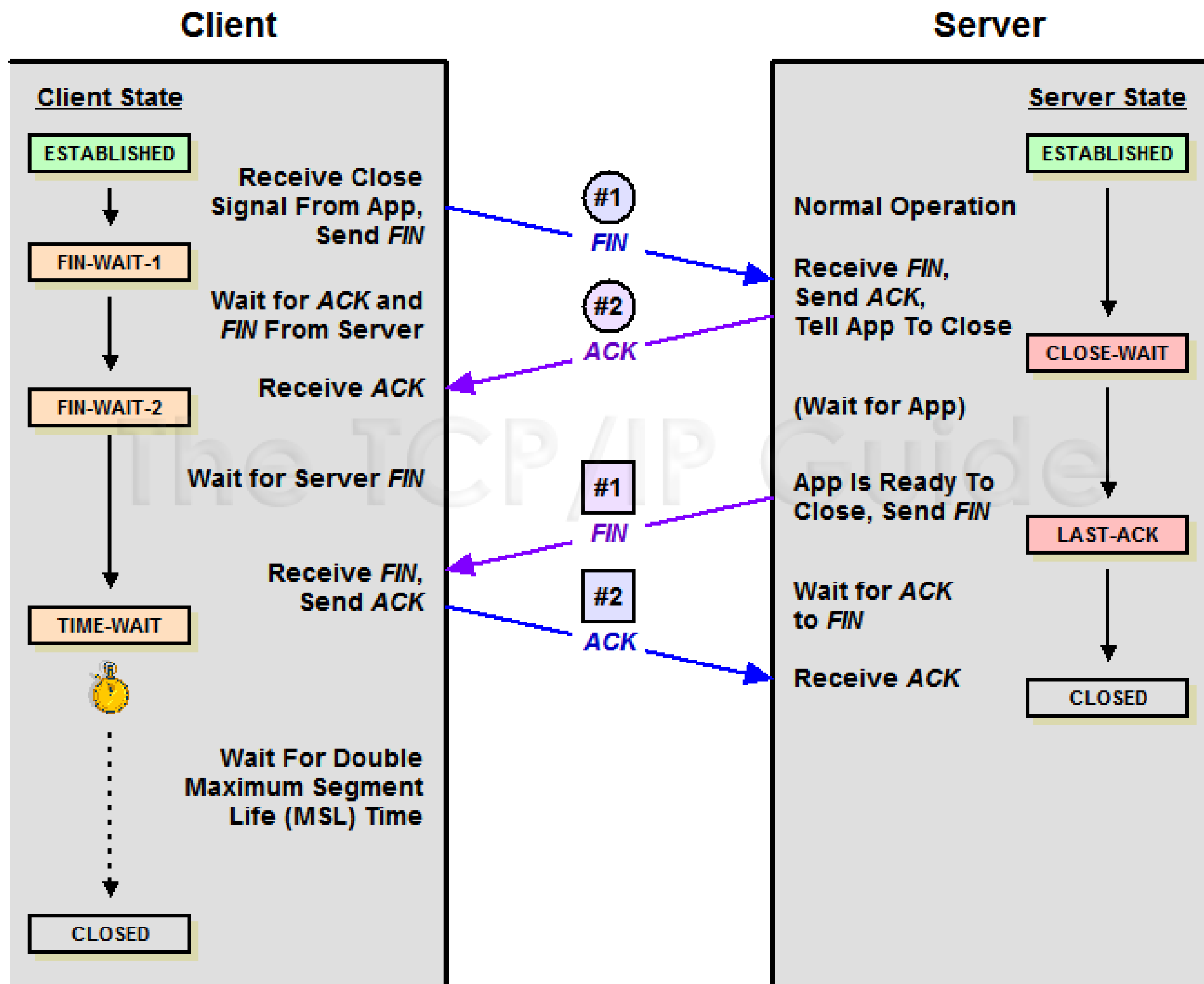


Figure 3.3: TCP Connection Termination[3]

2. Data transmission
3. Connection termination (see figure 3.3)

To achieve all goals, TCP requires the following information in each packet:

- **Sequence number(seq)** is used to specify the order of all packets.

[3] http://www.tcpipguide.com/free/t_TCPConnectionTermination-2.htm

- **Acknowledgement number(ack)** is calculated by the receiver. The sender determines if the receiver has received the data successfully by checking the acknowledgement number in the response.
- **Flags** (e.g., SYN, SYN-ACK) are used to specify the states of connections and packets.
- **Checksum** is used to prevent accidental mistakes in transmissions.

State Maintenance

To maintain the connection states required by TCP, we build a mapping relationship between one `TCPForwarder` and one TCP connection. Since a TCP connection can be determined by the IP address (the address is the same for all apps) and the port number, the relationship is based on the port number.

Implementation Details

The changes of states in TCP can be considered as a state machine. To model the state machine, we define several states:

- `LISTEN`: The connection is not established yet. The `TCPForwarder` is expecting a `SYN` packet.
- `SYN_ACK`: The `TCPForwarder` has received a `SYN` packet already from the application and responded to it with a `SYN-ACK` packet. The `TCPForwarder` is expecting an `ACK` packet to finish the connection establishment.
- `DATA`: The connection is established, and the `TCPForwarder` is transmitting data.
- `HALF_CLOSE_BY_CLIENT`: The client has sent out a `FIN` packet.
- `HALF_CLOSE_BY_SERVER`: The server has already responded with a `FIN` packet.
- `CLOSED`: The connection is closed.

We implement the state machine in the `TCPForwarder` as shown in figure 3.4. The following explains the implementation through a typical TCP connection life cycle:

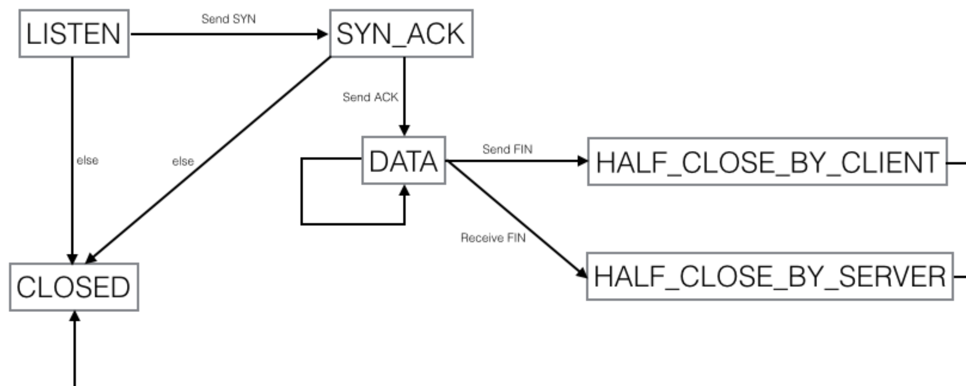


Figure 3.4: TCP state machine

1. When a TCPForwarder is initialized, it is in the LISTEN state and waiting for a 3-way handshake. When the TCPForwarder receives a SYN packet from an application, it will respond with a SYN_ACK packet to the application and go into the SYN_ACK state.

2. After receiving the SYN_ACK packet, the application will send an ACK packet, and the TCP connection is established. The TCPForwarder will then go into the DATA state after receiving that ACK packet. Also, the TCPForwarder will connect to the LocalServer.

3. In the DATA state, the TCPForwarder transmits each packet received from the Dispatcher to the LocalServer and responds with an ACK packet to the application. If the

LocalServer sends anything back, the TCPForwarder will also transmit it to the application with an appropriate sequence number.

4. Whenever the application sends a FIN packet, the TCPForwarder goes to the HALF_CLOSE_BY_CLIENT state, does some termination work and then goes to the CLOSED state.

5. Whenever the TCPForwarder receives a FIN packet from the LocalServer, it goes to the HALF_CLOSE_BY_SERVER state, does some termination work and then goes to the CLOSED state.

Connection Status

The TCP connection states (e.g., seq, ack) are stored in TCPConnectionInfos. The TCPConnectionInfo class implements all necessary methods to read and update the states.

Non-blocking Transmission

Because TCP is a stream delivery service, one request of another protocol using TCP (e.g., HTTP) may be divided into different TCP packets. In this case, using blocking transmission would be complex and introduce higher performance overhead. In our implementation, we use two threads. One thread sends requests to the LocalServer and the other reads responses from the LocalServer when it is readable.

TCP Checksum Recalculation

The TCP checksum is used to provide the error-checked feature. The checksum calculation requires the entire content of the TCP packet and part of the IP header. Details can be found online[4].

3.6.2 Connection with LocalServer

Each TCPForwarder communicates with the LocalServer on behalf of an application running on the device. The messages transferred between these two classes are requests (sent by the application) from the FakeVPNService and responses from real servers.

[4] http://en.wikipedia.org/wiki/Transmission_Control_Protocol#Checksum_computation

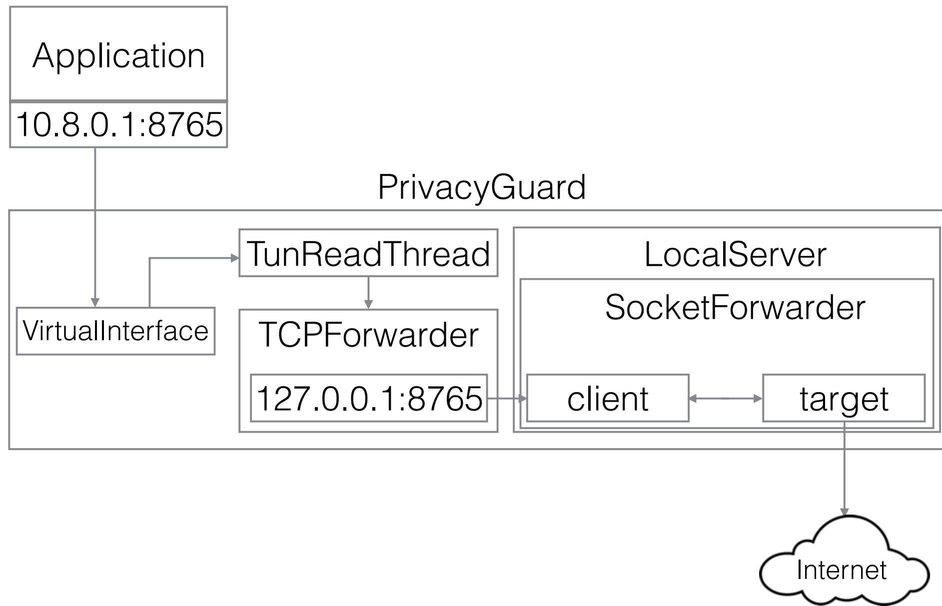


Figure 3.5: Configuration of the Sockets Used in PrivacyGuard

Each `TCPForwarder` connects to the `LocalServer` like connecting to a normal server. The `TCPForwarder` creates a socket and connects to the `LocalServer`. The socket binds to `127.0.0.1` and the source port in the TCP packets, i.e., the port number used by the corresponding application. Although the port numbers used by the application and that socket are the same, the IP addresses are different (the socket uses `127.0.0.1`, and the application uses `10.8.0.1`). Figure 3.5 shows this configuration. The reason we use the same port number is described in section 3.7.2.

There are two directions for the transmissions between the `TCPForwarder` to the `LocalServer`. These two directions are different.

Transmissions from `TCPForwarder` to `LocalServer`

In this direction, the `TCPForwarder`s retrieve the TCP data from IP datagrams and then send the data with the socket.

Transmissions from `LocalServer` to `TCPForwarder`

Since `TCPForwarders` and the `LocalServer` communicate with sockets, the data read from these sockets are payloads of TCP packets instead of IP datagrams required by `TunWriteThread.write()` as arguments(see section 3.5.2). We need to create an IP datagram by using the data and the status we maintain in `TCPConnectionInfos` (see section 3.6.1) and `TCPForwarders`. The TCP checksum recalculation is also necessary.

3.7 `LocalServer`

From the point of view of applications, the `LocalServer` is the real server. Applications only communicate with the `LocalServer`. The `LocalServer` listens on a specific port, which is known and connected to by all `TCPForwarders`.

3.7.1 Connection Accepted

When a "client" (in our case, a `TCPForwarder`) connects to the `LocalServer`, the `LocalServer` creates two sockets: `target` and `client`. Then the `LocalServer` creates a `SocketForwarder` (described in section 3.8) with these two sockets.

3.7.2 How to Determine the Real Server

The socket used by a `TCPForwarder` to connect to the `LocalServer` uses the same port number as the corresponding application (but the IP address of the socket is different, see section 3.6.2). We can use the same port number to determine to which IP address and port number the application actually wants to connect. In Linux and therefore also in Android, this information can be found in `/proc/net/tcp` or `/proc/net/tcp6` files (see appendix D for examples of these files). With these files, we can also determine the UID of the application that issues this connection. The application name can be obtained with the UID as well.

3.8 `SocketForwarder`

`SocketForwarders` match one to one with TCP connections. Each `SocketForwarder` contains two sockets, the `client` socket and the `target` socket. These two sockets are created

by the `LocalServer` for one particular connection (see section 3.7). A `SocketForwarder` contains a pair of threads:

- Outgoing Thread: This thread receives data from the `TCPForwarder` through the `client` socket. It forwards the data to the actual server through the `target` socket. Intuitively, since there cannot be privacy leakage in incoming data, the filtering is only done on messages from the `client` socket to the `target` socket. Because the current filtering approach is simple string matching, which can take much time if the message is long, we provide two options.
  1. Synchronous Filtering: filtering after reading data from the `client` socket and before sending to the `target` socket. With this option, PrivacyGuard can provide real-time protection and allow modifications to data.
  2. Asynchronous Filtering: adding data to a queue after reading data from the `client` socket and sending to the `target` socket right away. Another thread retrieves data from the queue and does the filtering. With this option, there is less network overhead caused by filtering, but we can detect leaks only retroactively.
- Incoming Thread: This thread receives data from the actual server through the `target` socket. It forwards the data to the `TCPForwarder` through the `client` socket.

3.8.1 Handling SSL Connections

As mentioned in section 3.2.4, the analysis requires plain text. However, SSL connections encrypt network traffic. If a client wants to establish an SSL connection with a server, the client and the server have to do an SSL handshake. In this handshake, the client receives a certificate from the server and negotiates the encryption key for encrypting all following messages. The pre-master-secret required to compute the encryption key is encrypted with the public key of the server contained in the certificate[5]. To obtain the plain text of the following messages, we need to obtain the corresponding private key of the public key to retrieve the pre-master-secret. However, deriving the private key from the public key in the SSL protocol is infeasible. To reveal the data, we need to deploy a man in the middle proxy.

Man in the Middle Proxy

[5] more details can be found in http://en.wikipedia.org/wiki/Transport_Layer_Security#TLS_handshake

[6] The picture is from http://www.secureworks.com/cyber-threat-intelligence/threats/transitive-trust/

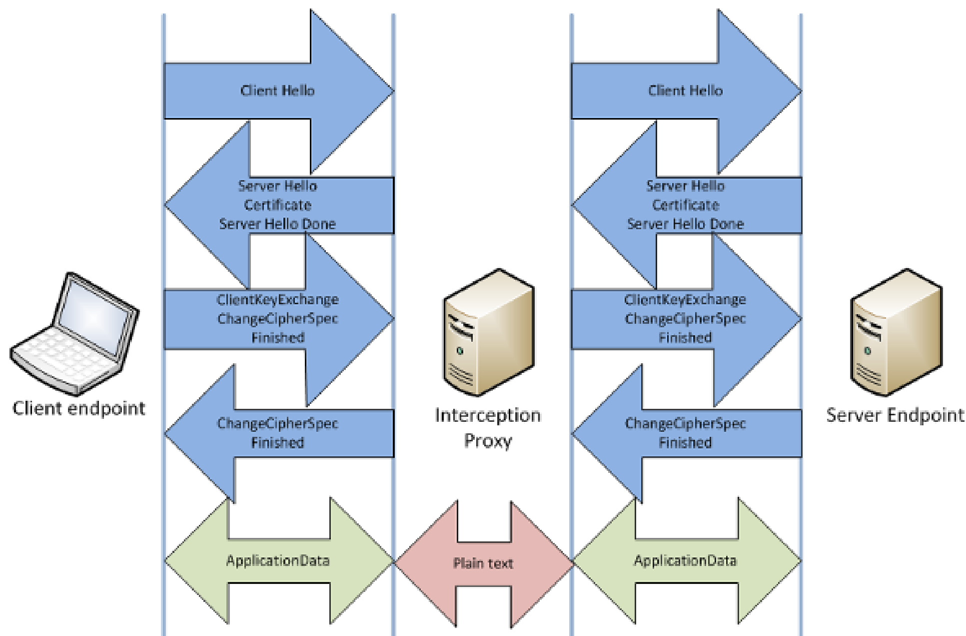


Figure 3.6: Man in the Middle Proxy[6]

As shown in figure 3.6, the proxy pretends to be the server in view of the client and pretends to be the client in view of the server. When a client wants to access the server, the proxy will send the client its own certificate and also connect to the real server. After the handshake, all messages from the client are decryptable to the proxy since the public-private key pair belongs to the proxy. For every message from the client, the proxy retransmits it to the server after maybe making some modifications.

Implementation of the Man in the Middle Proxy

`PrivacyGuard` creates a root CA certificate and installs the certificate on the device when `PrivacyGuard` launches the first time. Every certificate signed by this root CA certificate is then trusted by applications.

When an application wants to establish an SSL connection with a remote server, the `LocalServer` will first do a hostname lookup for the destination IP address of the remote server. In the lookup, the `LocalServer` tries to establish an SSL connection with the destination IP itself. During the establishment, the `LocalServer` can obtain the certificate of the destination server and retrieve necessary information about the server from the

certificate, such as the subject name. The `LocalServer` then creates fake certificates with this information for each website visited by applications with SSL connections. These certificates are signed by the root CA certificate. With fake certificates, the `LocalServer` can finish the SSL handshake with applications. Both the `client` and `target` sockets are also replaced with `SSLSocket`s. Although the traffic between applications and the `client` socket or between the `target` socket and the Internet is encrypted, we can read data from the `client` and `target` sockets in plain text. The source code for certificate generation and hostname lookup is from SandroProxy[7].

3.9 Customized Plugins

All plugins must implement the `IPlugin` interface (details of the interface can be found in appendix C). Since the `LocalServer` has implemented a man in the middle proxy, all network traffic between the `client` and `target` sockets is in plain text (see section 3.8.1). Corresponding methods of all plugins will be called to filter this traffic. With plugins, developers can easily set up their own analysis tools.

3.9.1 Implemented Plugins

So far, we have implemented three plugins:

- LocationDetection: detects location data. To detect if a message contains the current location, the plugin needs to obtain the location data itself. However, invoking `getLastKnownLocation()` for each message takes too much time. Instead, we register a `LocationListener` for every available location provider (The location provider is a concept of Android. Developers can obtain location data from available location providers).

- PhoneStateDetection: detects IMEI, IMSI, AndroidID, and user's own phone number.

- ContactDetection: detects email addresses and phone numbers by using regular expressions.

[7] https://code.google.com/p/sandrop/

3.10 Limitations

PrivacyGuard has some limitations:

- Encryption of sensitive information by an application can prevent detection since our current plugins filter with string matching. In 53 applications evaluated, we observe that there are three applications where we suspect that these three applications encrypt the data.

- Although our man-in-the-middle proxy works well in most situations, it cannot address the SSL pinning technique. If an application uses SSL pinning, the application will store the certificates it trusts locally. For any certificate this application receives while establishing an SSL connection, it rejects the certificate unless the certificate is one of these certificates installed locally. No man-in-the-middle proxy, including our implementation, works against SSL pinning unless the proxy modifies the certificates saved by the application. So far, we observe that only the Twitter[8] application uses SSL pinning.

- Currently, our man-in-the-middle proxy only supports RSA because we use SandroProxyLib[9]. This library only generates RSA key pairs.

- PrivacyGuard cannot distinguish legitimate sharings of sensitive data from illegitimate sharings. This limitation is also a common problem of other work. One possible solution is using Bayes classification [18]. With Bayes classification, we can compute how similar the data sent out is with the original sensitive data and determine whether the sharing is legitimate based on the similarity. Inspired by LP-Guardian [9], we can also use the host or IP address information to distinguish between legitimate and illegitimate sharings.

- TaintDroid can track data from gyroscopes, cameras and microphones. PrivacyGuard cannot detect this kind of data.

[8] https://play.google.com/store/apps/details?id=com.twitter.android

[9] https://code.google.com/p/sandrop/source/browse/projects/SandroProxyLib/src/org/sandrop/webscarab/plugin/pro

Chapter 4

Evaluation

To evaluate PrivacyGuard, we focus on two aspects: performance and effectiveness. For performance, we use UpDownLoader, a dummy application developed by ourselves, and SpeedTest.net[1] to test the network performance and power consumption. For effectiveness, we use a set of applications to train PrivacyGuard and use another set of applications to test whether PrivacyGuard can detect privacy leakages.

For effectiveness, we compare our results with TaintDroid since TaintDroid is a popular state-of-the-art taint-based privacy detection tool. It has been used for comparison in some other work ([18], [17]). For performance, we do not compare with TaintDroid since TaintDroid has 32% performance overhead on a CPU-bound micro-benchmark ([8]). This high overhead introduces much higher overhead in battery life than PrivacyGuard.

4.1 Performance Evaluation

For the performance evaluation, we mainly consider two parts: network performance and power consumption.

4.1.1 Setup

We set up three programs:

[1]https://play.google.com/store/apps/details?id=org.zwanoo.android.speedtest&hl=en

- UpDownLoader: an Android application. It has two functionalities:
  1. upload and download a file - used for evaluating network performance
  2. keep uploading or downloading for a specified period - used for evaluating power consumption
- DummyServer: a desktop program. It runs on a desktop and cooperates with UpDownLoader.
- SpeedTest.net: an Android application for testing network speed and ping delay.

4.1.2 Network Performance

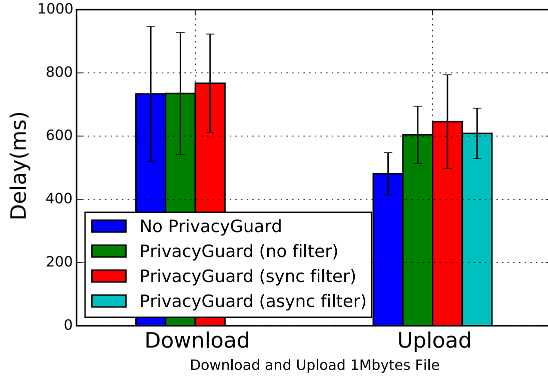
Section 3.9 shows that filtering is done by string matching. String matching may take much time, especially if the network message is long. Many memory allocations as well as string operations, such as concatenation or copy, are executed. These operations cost time and thus may cause a network delay since messages are sent out after the filtering in the synchronous configuration. Furthermore, there is one additional network I/O operation between the `TCPForwarder` and the `LocalServer`, which may increase the latency. This evaluation measures how PrivacyGuard affects the network performance (both the latency and the throughput).

Four scenarios are tested: no PrivacyGuard running, PrivacyGuard without filtering privacy leakages, PrivacyGuard with synchronous filtering, and PrivacyGuard with asynchronous filtering. For each of the first three scenarios, separate experiments were executed with different settings described below. For the fourth scenario, since the asynchronous filtering does not affect the download speed, we only test it with uploading files.

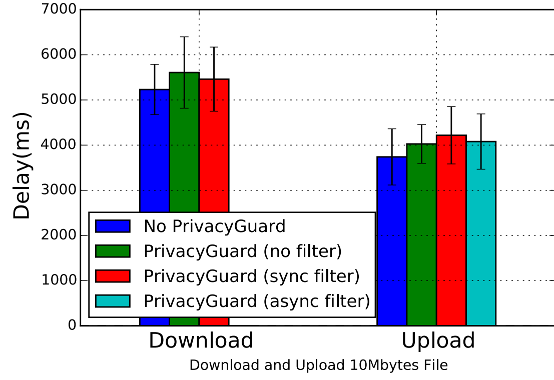
Experiment Settings

- Download and upload one 1 Mbytes file
- Download and upload one 10 Mbytes file
- Test with SpeedTest.net (ping delay, download speed, upload speed)

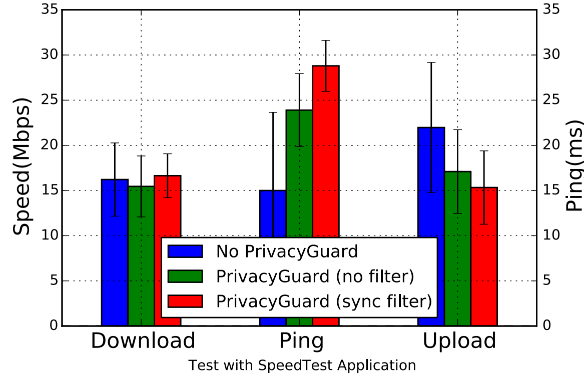
In the first two settings, we use UpDownLoader. It simply downloads and uploads files from a server. The method we use to calculate the delay is `System.currentTimeMillis()`.



(a) Result of 1 Mbytes File Experiments



(b) Result of 10 Mbytes File Experiments



(c) Result of SpeedTest.net Experiments

Figure 4.1: Results of Network Performance Evaluations

Methodology

Because we use the Wi-Fi network to run these experiments, we run the same experiment 10 times for each scenario-setting pair to address potential instability. Also, we test each scenario one after another to make sure the network conditions are similar on the average.

Analysis

The experiment results are shown in figure 4.1. All figures show the average values and the standard deviations. The original data can be found in appendix A.1.

For the downloading part of figures 4.1a, 4.1b, 4.1c, we can see that the downloading delay is similar between different scenarios. This observation is as expected because as described in section 3.8, PrivacyGuard does not filter the incoming network traffic and thus does not introduce much overhead.

For the uploading part, we can see that PrivacyGuard (with sync filtering) is 34.3%(164.9ms) and 12.81%(479.1ms) slower than no PrivacyGuard in the 1Mbytes and 10Mbytes experiments, respectively. PrivacyGuard (with async filtering) is 26.55%(127.7ms) and 9.07%(339.3ms) slower than no PrivacyGuard in the 1Mbytes and 10Mbytes experiments, respectively. For PrivacyGuard without filtering, it is 25.64%(123.3ms) and 7.67%(286.8ms) slower than no PrivacyGuard running in the 1Mbytes and 10Mbytes experiments. In conclusion, filtering does not introduce much overhead. Furthermore, from the result, we can see that the relative overhead in uploading gets lower as the size of the file increases.

For the experiments with SpeedTest.net, we can see that the downloading results are similar for all three settings. For the ping delay, PrivacyGuard introduces about 59.33% overhead without filtering and 92.0% overhead with sync filtering. Although the percentage is high, the absolute increase of the ping delay remains acceptable. PrivacyGuard's ping delay is 26ms in average, only 11ms longer than no PrivacyGuard. For the uploading speed, PrivacyGuard without filtering is 22.19% slower than no PrivacyGuard and PrivacyGuard with filtering is 30.22% slower. While doing the experiments, we observed that the SpeedTest.net application reached high upload speeds at the beginning of an experiment with PrivacyGuard running, but there were sometimes EPIPE errors later in the experiment. This error often means that the other end of the pipe no longer exists and the network connection is broken. As we observed, the SpeedTest.net application opens multiple TCP connections for uploading or downloding. EPIPE errors would shut down some of the TCP connections and thus slow down the speed. Also, from table A.3, we can see that PrivacyGuard achieves similar uploading speeds as the scenario without PrivacyGuard running in some tests. We analyzed the TCP packets to see if PrivacyGuard sends any invalid packets that cause this error, but we could not find any. We also tested with tPacketCapture[2], a close-source application that uses `VPNService` for packet capture, and also observed slow uploading speeds. PrivacyGuard should be able to achieve higher speeds once we figure out the reason for this issue.

[2]https://play.google.com/store/apps/details?id=jp.co.taosoftware.android.packetcapture&hl=en

Although PrivacyGuard causes overhead in the uploading tasks, this large-content and high-frequency uploading is rare in daily use. Usually, the outgoing packets are short HTTP requests. The latency increase remains acceptable. PrivacyGuard can achieve high throughput because the filtering takes less time when the packet is small.

4.1.3 Power Consumption

We consider three scenarios: no PrivacyGuard running, PrivacyGuard without filtering, PrivacyGuard with synchronous filtering. For each scenario, we use UpDownLoader mentioned above to run separate experiments for downloading and uploading. This application downloads or uploads a 1Mbytes file every 10 seconds for a specified period. The method we use to retrieve the battery level is:

```
// return the percentage of battery left
double getBatteryLevel() {
  // get the status of battery
  Intent batteryStatus = this.registerReceiver(
    null,
    new IntentFilter(Intent.ACTION_BATTERY_CHANGED)
  );
  // EXTRA_LEVEL returns level of battery left
  // EXTRA_SCALE returns the highest level of batter
  return 1.0 * batteryStatus.getIntExtra(BatteryManager.EXTRA_LEVEL, -1) /
               batteryStatus.getIntExtra(BatteryManager.EXTRA_SCALE, -1);
}
```

Methodology

For each experiment, we follow these steps:

1. Fully charge the device
2. Connect to the DummyServer running on a desktop
3. Start downloading or uploading a 1Mbyte file every 10 seconds for 60 minutes
4. Turn off the screen
5. The device will vibrate and record the decrease in battery level to a file when the task finishes

Table 4.1: Decrease in Battery Level for Power Consumption Experiments

Settings	Upload	Download
No PrivacyGuard	3.00%	3.99%
PrivacyGuard without filtering	3.00%	4.00%
PrivacyGuard with sync filtering	3.00%	4.00%

Result

The result is shown in table 4.1. From the table, we can see that PrivacyGuard introduces almost no power consumption even with a high frequency of uploading and downloading. This result is reasonable since we observe no significant increase on the CPU load by PrivacyGuard. We are confident that it is practical to use PrivacyGuard in daily life.

4.2 Effectiveness Evaluation

In this part, we evaluate whether PrivacyGuard can detect privacy leakages of applications effectively.

4.2.1 Setup

We test both TaintDroid and PrivacyGuard on both emulators and a real device.

- Emulator: we set up two emulators. One uses TaintDroid 4.3r_1 (which is the most recent version of Taintdroid), and the other uses stock Android 4.3r_1 with PrivacyGuard.
- Device: we use a Nexus 4 that runs both TaintDroid 4.3r_1 and PrivacyGuard.

We use both emulators and a device for following reasons:

- Some components of the Google Play service required by some applications are missing in the emulators.
- Some features are not available in the emulators, such as the network location provider.

Initially, we wanted to run all experiments on the emulators because we only have one device and it is easier to automate experiments on the emulator. We tried to set up the Google Play service required by some applications. Although the Android virtual device manager provides emulators with Google API support, we were not able to have the TaintDroid image support the Google API in the same way. Instead, we downloaded Google framework apps from Goo.im[3] and pushed these to the emulators. The framework is not complete, but some apps can now run on the emulators. The setup script can be found in appendix B.

Due to the remaining difficulties, such as the incomplete Google Play service and the missing features, we also use a real device. Since we only have one device, we run both TaintDroid and PrivacyGuard on it. Because TaintDroid by default refuses to execute applications that use third-party native libraries, we modified Taintdroid to have it execute this kind of apps. This modification is also necessary for PrivacyGuard because it requires a native library from SandroProxy for hostname lookup.

4.2.2 Methodology

The applications we use to train and test have been used in some other papers ([18], [9]). In this way, we can compare our results with theirs. We use 13 out of 15 applications from table 3 of Tripp and Rubin [18] since the other two are not available from Google Play anymore. For the applications used by Fawaz et al. [9], we asked them for their list. Also, there are several applications in the list that are not available anymore.

In the training phase, we use the output of TaintDroid as the ground truth to derive filters to be used by PrivacyGuard. In the testing phase, we use these filters to filter network traffic and compare our result with TaintDroid's result.

For an evaluated application, we do the following:

1. We first uninstall the application if it is already installed, and then install it. We do this to clear all caches.

2. Before using the app, we first use the monitor tool provided by Android to set up the GPS location provider of the emulators and then launch our simpleLocation application. This app simply reads location values from the provider such that the `getLastKnownLocation()` method can return a valid location to the evaluated application later. Running this app to make sure `getLastKnownLocation()` returns a

[3] https://goo.im/

Table 4.2: Training Apps List

App	Category	TaintDroid			PrivacyGuard		
		Dev.ID	Location	Contact	Dev.ID	Location	Contact
com.yelp.android	Travel&Local		✓			✓	
com.yahoo.mobile.client.android.weather	Weather		✓			✓	
com.shazam.android	Music&Audio	✓	✓		✓	✓	
com.weather.Weather	Weather		✓			✓	
com.groupon	Shopping	✓	✓		✓	✓	
com.staircase3.opensignal	Tools	✓	✓		✓	✓	
com.yellowpages.android.ypmobile	Travel&Local		✓			✓	
com.urbanspoon	Travel&Local	✓	✓		✓	✓	
com.twitter.android	Social		✓	✓			
com.aws.android	Weather		✓			✓	

valid value is necessary because some applications obtain the location only by calling that method, instead of registering a location update listener.

3. We manually deep crawl the evaluated app in the two emulators at the same time to make sure we execute the same operations. While crawling the app, we record all operations by using RERAN[4] such that we can replay all operations if we need to redo the experiments. Initially, we planned to use PUMA [13] for automatic crawling. However, we gave up for the following reasons: First, we were unable to figure out how to add PUMA scripts, and the code does not contain the examples shown in the paper. Also, the crawling approach PUMA has already included was not able to recognize all UI components. Furthermore, we still need to manually type the username and password required by some applications, such as Facebook.

4.2.3 Training

We use ten applications from our list of applications for training purposes. In this phase, PrivacyGuard simply records all network traffic. From the output of TaintDroid, we can find the type of information it detects and the traffic that contains the information. For each alert raised by TaintDroid for a specific application, we manually look for the corresponding data in the network traffic of this application recorded by PrivacyGuard. From the corresponding data, we read the traffic to find possible patterns and design a filter for the patterns. The applications we use for training are shown in table 4.2. The checkmark means the corresponding information is detected in the network traffic of that application.

The data we try to detect and the filters we find are in table 4.3. For the location

[4] http://www.androidreran.com/

Table 4.3: Filters

Data	Source	Filters
Location	all available location providers	keep two decimal points
Dev.ID	IMEI, IMSI, AndroidID	plain text, SHA, MD5
Contact	phone number, email address	regular expression

data (i.e., latitude and longitude), we keep two decimal points of the floating point values as filters. For the device identifiers, we also compute the SHA-1 and MD5 hashes of the original data as filters since some applications send these hashes instead. These filters may be incomplete, but it is easy to add more.

In the training phase, we observe that there are many network packets that are detected as leaks by Taintdroid but for which we fail to find obvious variants of private data in these packets. Therefore we could not derive appropriate filters for these packets. Some of them are links to pictures. We suspect that these leaks are false positives.

Also, while doing experiments on the real device, there are many SMS leakage notifications from TaintDroid for almost every application. We read the source code of TaintDroid and learn that TaintDroid will set the SMS taint flag if the `ContentProvider.query()` method finds that there is a `sms://` or `mms://` string in the given URI for querying. However, we cannot find any of these two strings in the decompiled source code of these applications. What makes us even more confused is that these apps do not have any SMS related permissions in their `AndroidManifest.xml` files.

4.2.4 Testing Results

For PrivacyGuard, we use the filters from the training phase to analyze network traffic. For TaintDroid, we run it and record its notifications. The testing results are shown in table 4.4 and table 4.5.

We also run PrivacyGuard and TaintDroid on some of the most popular advertising libraries. For each library, we develop a dummy application to wrap it. As shown in table 4.5, PrivacyGuard works for four out of five libraries and TaintDroid fails on Amazon's library. The reason for the failure of TaintDroid is that Amazon's library uses native code to round up floating point numbers.

Table 4.6 shows how many applications have privacy leakages detected by TaintDroid and PrivacyGuard.

Table 4.4: Testing Results of Applications

App	Category	TaintDroid			PrivacyGuard		
		Dev.ID	Location	Contact	Dev.ID	Location	Contact
com.google.android.apps.maps	Travel&Local					✓	
com.facebook.katana	Social		✓			✓	
com.android.chrome	Communication		✓			✓	
com.google.android.apps.plus	Social		✓			✓	
fr.epicdream.beamy	Shopping	✓	✓		✓	✓	
net.flixster.android	Entertainment		✓			✓	
org.zwanoo.android.speedtest	Tools	✓	✓		✓	✓	
com.imdb.mobile	Entertainment	✓			✓	✓	
gbis.gbandroid	Travel&Local		✓		✓	✓	
com.zc.android	Transportation					✓	
org.wikipedia	Books&Reference		✓			✓	
com.starbucks.mobilecard	Lifestyle		✓				
com.joelapenna.foursquared	Travel&Local		✓			✓	
com.ikea.app	Lifestyle						
thecouponsapp.coupon	Shopping		✓		✓	✓	
com.magnifis.parking	Transportation		✓		✓		
com.levelup.beautifulwidgets.free	Personalization						
com.chrome.beta	Productivity		✓			✓	
com.fitnesskeeper.runkeeper.pro	Health&Fitness		✓			✓	
ch.search.android.search	Books&Reference					✓	
org.mozilla.firefox	Communication					✓	
com.evernote.food	Lifestyle		✓			✓	
com.microsoft.bing	Books&Reference					✓	
com.walmart.android	Business		✓		✓	✓	
com.webmd.android	Health&Fitness		✓			✓	
com.antivirus	Communication	✓			✓		
com.appshop.ios7lockscreen_2	Personalization						
com.bestcoolfungames.antsmasher	Game/Arcade				✓		
com.cleanmaster.mguard	Tools				✓		
com.coolfish.cathairsalon	Game/Casual				✓		
com.digisoft.TransparentScreen	Entertainment		✓		✓	✓	
com.g6677.android.cbaby	Game/Casual	✓			✓		
com.g6677.android.chospital	Game/Casual	✓			✓		
com.g6677.android.design	Game/Casual	✓			✓		
com.g6677.android.pnailspa	Game/Casual	✓			✓		
com.g6677.android.princesshs	Game/Casual	✓			✓		
com.goldtouch.mako	News&Magazines				✓		
com.dictionary.com	Books&Reference	✓			✓		

Table 4.5: Testing Results of Advertising Libraries

App	Category	TaintDroid			PrivacyGuard		
		Dev.ID	Location	Contact	Dev.ID	Location	Contact
admob	Ad Library						
amazon	Ad Library					✓	
airpush	Ad Library		✓			✓	
inmobi	Ad Library		✓			✓	
mopub	Ad Library		✓			✓	

Table 4.6: Testing Results Analysis

Number of Applications Detected	TaintDroid	PrivacyGuard
Location	21	26
Device ID	10	19
Contact	0	0

For the device IDs, the reason TaintDroid fails is that it does not detect IMEI leaks. We do not know why TaintDroid is unable to detect IMEI leaks.

For the location, we can see that PrivacyGuard can detect more leaks than TaintDroid. However, there are some applications for which PrivacyGuard is unable to detect leakages, such as *com.starbucks.mobilecard* or *com.dictionary.com*. They use the Google map API for location-based services. Inspecting the network traffic indicates that this API seems to obfuscate the location data. The obfuscation makes it almost impossible for PrivacyGuard to detect the leakage. This limitation is also mentioned in section 3.10.

4.3 Conclusion

With the result of our evaluations, we can conclude that PrivacyGuard introduces acceptable overhead on the network performance and almost no overhead on the battery life. It is possible to use PrivacyGuard in daily life. Also, by using the filters we designed based on the analysis of network traffic in the training phase, PrivacyGuard can detect almost all leakages from all applications.

Chapter 5

Future Work

There is a great deal of possible future work. We list some topics below.

Privacy Enforcement Algorithms

Since PrivacyGuard provides access to the plain network traffic, it is possible to implement anonymity algorithms. These algorithms can coarsen the private data in the network traffic without affecting the usability of applications. PrivacyGuard provides an easier way for prototyping these strategies than modifying Android ([9], [14]). Also, because PrivacyGuard can do the filtering before sending out a message, it is possible to ask the user for confirmation before sending the message.

Advanced Filters

Currently, the filtering plugins execute a simple string search to detect privacy leakages. This simple string searching can be inadequate. Inspired by Tripp and Rubin ([18]), we can improve the detection rate and reduce the false positive rate by adding Bayes classifications. In general, it is possible to implement many classification algorithms to make the detection more accurate.

Destination-Aware Filters

Although it is difficult to distinguish network traffic issued by advertising libraries from network traffic generated by the core of applications in PrivacyGuard, we can leverage

the destination information to distinguish. The hostname or the IP address can help us. For example, we can add a black list of IP addresses where we do not want any sensitive information to be sent.

Location Based Filtering Policies

Location-based filtering is easy to add in PrivacyGuard. For example, it could be useful to give users the option to block location data only in a specified period, in specified places or for some applications.

Bug Fixes and Code Optimization

During the experiments, we observed that there are some EPIPE errors, which dramatically slow down the uploading speed. From table A.3 for the network performance experiments, we can see that the uploading speed can reach a high value in some tests. We could not find the reason, but there is a potential improvement in the uploading speed.

User-friendly Interfaces

PrivacyGuard provides a platform for developers to develop their own plugins. These plugins may need proper configurations to adapt to different requirements of different users. Also, there may be too many notifications with many plugins installed. Adding user-friendly interfaces could help users configure their plugins as well as avoid distraction.

Chapter 6

Conclusion

We propose and implement a new approach, PrivacyGuard, to detect privacy leakages. PrivacyGuard does not require root permissions and is portable to all devices with Android versions newer than 4.0. It is easy to use without any knowledge about security or privacy. It is extensible and configurable, and it provides a new option for prototyping other privacy enforcement algorithms. With the support of a man-in-the-middle proxy, PrivacyGuard can also filter SSL traffic if there is no SSL pinning. In addition, it introduces acceptable overhead in both the power consumption and the network performance. PrivacyGuard is practical to use in daily life. According to the experiment results, PrivacyGuard can efficiently detect privacy leakages of most applications.

PrivacyGuard is open-source and available in Bitbucket[1].

[1] https://bitbucket.org/Near/locationguard

APPENDICES

Appendix A

Experiment Data

A.1 Network Performance Experiments

Table A.1: Experiment Data for 1Mbytes file experiments

(ms)	No PrivacyGuard		PrivacyGuard (no filter)		PrivacyGuard (sync filter)		PrivacyGuard (async filter)	
	Up	Down	Up	Down	Up	Down	Up	Down
1	418	563	578	640	563	593	604	N/A
2	425	756	739	515	574	646	553	N/A
3	512	539	532	680	600	757	572	N/A
4	546	569	558	625	541	626	534	N/A
5	618	838	612	742	585	884	608	N/A
6	495	891	596	750	930	946	738	N/A
7	449	1224	793	1213	533	851	520	N/A
8	410	561	524	777	904	938	750	N/A
9	504	759	561	823	672	896	637	N/A
10	432	635	549	583	556	535	570	N/A
Avg	480.9	733.5	604.2	734.8	645.8	767.2	608.6	N/A
Overhead			25.64%	0.18%	34.3%	4.60%	26.55%	N/A

Table A.2: Experiment Data for 10Mbytes file experiments

(ms)	No PrivacyGuard		PrivacyGuard (no filter)		PrivacyGuard (sync filter)		PrivacyGuard (async filter)	
	Up	Down	Up	Down	Up	Down	Up	Down
1	3372	4804	3982	6115	4509	6490	4828	N/A
2	4631	4871	4871	5116	4041	4929	4456	N/A
3	3641	5707	3847	5218	3898	5078	3366	N/A
4	3299	5675	4273	7046	5406	5168	4948	N/A
5	4663	5920	3945	5721	4549	6386	4165	N/A
6	4578	5672	4377	6751	3800	5321	3328	N/A
7	3229	5688	3780	4797	3455	4554	3748	N/A
8	3204	4377	3224	4852	3718	5264	3788	N/A
9	3353	4623	3913	5419	3769	4926	3513	N/A
10	3425	4970	4051	5038	5041	6451	4648	N/A
Avg	3739.5	5230.7	4026.3	5607.3	4218.6	5460.3	4078.8	N/A
Overhead			7.67%	7.20%	12.81%	4.39%	9.07%	N/A

Table A.3: Experiment Data for SpeedTest.net

Ping(ms) Up/Download(Mbps)	No PrivacyGuard			PrivacyGuard (no filter)			PrivacyGuard (sync filter)		
	Ping	Down	Up	Ping	Down	Up	Ping	Down	Up
1	13	17.39	22.89	19	17.11	20.81	29	15.26	19.04
2	12	19.35	20.71	29	18.41	21.52	25	20.05	21.43
3	14	16.92	18.69	29	15.62	19.03	30	17.05	14.55
4	12	17.72	27.48	25	17.65	23.81	25	18.82	12.34
5	13	18.29	27.60	21	16.58	13.68	25	16.31	13.15
6	39	20.21	27.48	22	19.07	21.02	30	13.69	13.77
7	16	6.84	3.26	25	7.88	11.34	33	13.58	11.82
8	12	19.01	24.24	20	11.67	12.50	31	17.23	13.25
9	10	12.95	24.07	29	15.27	11.89	30	14.41	11.57
10	9	13.49	23.30	20	15.31	15.37	30	19.99	22.40
Average	15	16.217	21.972	23.9	15.457	17.097	28.8	16.639	15.332
Overhead				59.33%	-4.69%	-22.19%	92.0%	2.60%	-30.22%

Appendix B

Script for Setting up Google Play Service on Emulators

```
adb -s $device shell mount -o remount r,w /system
adb -s $device shell chmod 777 /system/app
adb -s $device push "${GOOGLEPLAY_DIR}/GoogleLoginService.apk" /system/app/.
adb -s $device push "${GOOGLEPLAY_DIR}/GoogleServicesFramework.apk" /system/app/.
adb -s $device push "${GOOGLEPLAY_DIR}/Vending.apk" /system/app/.
adb -s $device shell rm /system/app/SdkSetup*
```

Appendix C

Interface for Plugins

```
package com.y59song.Plugin;

import android.content.Context;

public interface IPlugin {
  // filter on requests and responses.
  // our plugins only implement handlerequest now
  public String handleRequest(String request);
  public String handleResponse(String response);

  // change the request or response to achieve shadow or anonymity
  public String modifyRequest(String request);
  public String modifyResponse(String response);

  // set a context member to get access to device information
  public void setContext(Context context);
}
```

Appendix D

/proc/net/tcp Files Example

```
sl local_address rem_address st tx_queue rx_queue tr tm->when retrnsmt
uid timeout inode
0: 62CE140A:95A8 B7A7CDCB:1388 01 00000000:00000000 00:00000000 00000000 10102 0
1854718 1 00000000 310 4 30 10 -1
```

References

[1] Hazim Almuhimedi, Florian Schaub, Norman Sadeh, Idris Adjerid, Alessandro Acquisti, Joshua Gluck, Lorrie Cranor, and Yuvraj Agarwal. Cmu-isr-14-116 your location has been shared 5,398 times! a field study on mobile app privacy nudging.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, 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. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2014.

[3] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, Phillipa Gill, and David Lie. Short paper: a look at smartphone permission models. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 63–68. ACM, 2011.

[4] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[5] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguard–enforcing user requirements on android apps. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 543–548. Springer, 2013.

[6] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.

[7] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. *Mobile Security Technologies*, 2012.

[8] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, 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. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[9] Kassem Fawaz and Kang G Shin. Location privacy protection for smartphone users. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 239–250. ACM, 2014.

[10] Huiqing Fu, Yulong Yang, Nileema Shingte, Janne Lindqvist, and Marco Gruteser. A field study of run-time location access disclosures on android smartphones. *Proc. USEC*, 14, 2014.

[11] Marco Gruteser and Dirk Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 31–42. ACM, 2003.

[12] Hao Hao, Vicky Singh, and Wenliang Du. On the effectiveness of api-level access control using bytecode rewriting in android. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 25–36. ACM, 2013.

[13] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 204–217. ACM, 2014.

[14] Peter Hornyack, 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. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM, 2011.

[15] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2012.

[16] Christopher Mann and Artem Starostin. A framework for static detection of privacy leaks in android applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1457–1462. ACM, 2012.

[17] Julian Schutte, Dennis Titze, and JM De Fuentes. Appcaulk: Data leak prevention by injecting targeted taint tracking into android apps. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, pages 370–379. IEEE, 2014.

[18] Omer Tripp and Julia Rubin. A bayesian approach to privacy enforcement in smartphones. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 175–190, San Diego, CA, August 2014. USENIX Association.

[19] Philipp von Styp-Rekowsky, Sebastian Gerling, Michael Backes, and Christian Hammer. Idea: callee-site rewriting of sealed system libraries. In *Engineering Secure Software and Systems*, pages 33–41. Springer, 2013.

[20] Ge Zhong and Urs Hengartner. A distributed k-anonymity protocol for location privacy. In *Pervasive Computing and Communications, 2009. PerCom 2009. IEEE International Conference on*, pages 1–10. IEEE, 2009.