# PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices

Yihang Song and Urs Hengartner
Cheriton School of Computer Science
University of Waterloo
Waterloo, ON Canada
{y59song,urs.hengartner}@uwaterloo.ca

## ABSTRACT

More and more people rely on mobile devices to access the Internet, which also increases the amount of private information that can be gathered from people's devices. Although today's smartphone operating systems are trying to provide a secure environment, they fail to provide users with adequate control over and visibility into how third-party applications use their private data.

Whereas there are a few tools that alert users when applications leak private information, these tools are often hard to use by the average user or have other problems. To address these problems, we present PrivacyGuard, an open-source VPN-based platform for intercepting the network traffic of applications. PrivacyGuard requires neither root permissions nor any knowledge about VPN technology from its users. PrivacyGuard does not significantly increase the trusted computing base since PrivacyGuard runs in its entirety on the local device and traffic is not routed through a remote VPN server. We implement PrivacyGuard on the Android platform by taking advantage of the `VPNService` class provided by the Android SDK.

PrivacyGuard is configurable, extensible, and useful for many different purposes. We investigate its use for detecting the leakage of multiple types of sensitive data, such as a phone's IMEI number or location data. PrivacyGuard also supports modifying the leaked information and replacing it with crafted data for privacy protection. According to our experiments, PrivacyGuard can detect more leakage incidents by applications and advertisement libraries than TaintDroid. We also demonstrate that PrivacyGuard has reasonable overhead on network performance and almost no overhead on battery consumption.

## 1. INTRODUCTION

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. Since people carry their devices all day and use them to communicate or work, these devices can contain much private information. This information makes it possible to track, identify or profile users.

All popular mobile operating systems come with a mechanism to control access to this information by applications. The App Store has a review process to avoid malicious applications. iOS, Windows Phone, and upcoming Android M have a first-time use confirmation prompt for access to location and microphone among others. Users can also manually set the access policy for each application later in the settings. The current versions of Android use a permission-based security model to restrict applications from accessing private data and privileged resources. When users install an application, they are prompted to grant the permissions or cancel the installation. However, none of these approaches is satisfactory, especially when users do not understand what happens. Significant efforts have been made to explore these challenges [3][4][14].

The other problem with the approaches employed by existing platforms is that they are all-or-nothing. Users can control only whether an application can access a certain type of private data. Intuitively, there will be an information leakage only if the application also outputs this information, like in a network transmission. However, the existing mechanisms block access no matter how this data is used by the application and how it is processed before being transmitted. The mechanisms may block even if this data is not transmitted at all. For example, an application may access the location to obtain the current time zone, or it may implement an anonymization algorithm and send out a well-crafted location that does not leak any essential information. These all-or-nothing approaches therefore can affect the usability of benign applications.

Much existing work cannot address the problems described above. For example, some tools [2][8] leverage taint analysis, which sets a taint flag on the data returned by privacy-sensitive methods and checks if the data reaching a sink, such as the network, has the flag set. However, the taint will be propagated no matter how the data is processed by an application. Application-rewriting approaches [5] [7] [14], which modify an application to achieve better access control, also do not consider how applications use the private data. All these methods usually also have some other problems, such as the requirement of root permissions or of modifications to the Android kernel. In turn, these requirements make these approaches hard to use for the average user.


*SPSM'15,* October 12, 2015, Denver, Colorado, USA.

DOI: `http://dx.doi.org/10.1145/2808117.2808120`.

Most information leakage occurs over the network and could likely be detected by inspecting network traffic. We present PrivacyGuard, a VPN-based platform for intercepting the network traffic of applications. PrivacyGuard requires neither root permissions nor any knowledge about VPN technology from its users. PrivacyGuard does not significantly increase the trusted computing base since PrivacyGuard runs in its entirety on the local device and traffic is not routed through a remote VPN server. We implement PrivacyGuard on the Android platform by taking advantage of the `VPNService` class provided by the Android SDK.

PrivacyGuard can be used for many purposes. Apart from leakage detection, it could also be used to, for example, characterize the network traffic of an application, block advertisements, or filter cookies. We focus on its use for detecting the leakage of multiple types of sensitive data, such as a phone's IMEI number or location data. PrivacyGuard also supports modifying the leaked information and replacing it with crafted data for privacy protection. Our existing filters for detecting information leakage and our strategies for alerting users of a leakage are simple, but sufficient to serve as proof-of-concept. This makes PrivacyGuard an attractive platform for privacy researchers to explore more sophisticated filtering and user notification strategies [1].

We make the following contributions:

- We present the design of PrivacyGuard, which we believe to be the first open-source[1] VPN-based platform for Android that can intercept the network traffic of applications without requiring a remote VPN server.
- We describe the implementation of PrivacyGuard based on our design and the challenges we had to overcome.
- Our effectiveness experiments demonstrate that PrivacyGuard can detect information leakage in most applications effectively and has better detection results than TaintDroid [8], a popular taint-based approach.
- Our performance experiments show that PrivacyGuard introduces acceptable overhead in network performance and little overhead in battery consumption.

During the development of PrivacyGuard, we learned about an open-source proxy-based network interception tool for Android called SandroProxy[2]. (We discuss the disadvantages of a proxy-based approach in section 3.) We had several discussions with its author and benefited from his experience. We shared the source code of PrivacyGuard with him, and he provided advice about fixing problems in PrivacyGuard. PrivacyGuard uses some TLS interception code from SandroProxy (see section 4.5.2). The author of SandroProxy recently also added VPN-based network interception to his tool.

# 2. RELATED WORK

## 2.1 Taint Analysis

Taint analysis detects information leakage by analyzing applications. The analysis tool adds taint to the data returned by a source method and tracks how the taint propagates until it reaches a sink method. In general, there are two kinds of approaches: dynamic taint analysis and static taint analysis.

### 2.1.1 Dynamic Taint Analysis

Dynamic taint analysis monitors the data flow 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 propagation is added. Sink methods are also modified to check if the arguments, class members or static variables that these methods use are tainted. The advantages of dynamic taint analysis are that there is a report only when there is a data flow from a source to a sink and that it is relatively easy to use once set up. The disadvantages are false positives, where a leak may be reported even if the leaked data has been processed and is no longer sensitive or even if the data needs to be transmitted to provide an application's functionality. In addition, this approach requires root permissions, which introduces vulnerabilities, and requires re-flashing a device, which makes the approach less usable. Porting it to different devices or different Android versions may also be difficult. A popular dynamic taint analysis tool is TaintDroid [8]. Currently TaintDroid cannot handle applications that come with native libraries. There have also been some simple approaches to bypass TaintDroid [18]. BayesDroid [19] is an enhancement to TaintDroid and addresses the problem with false positives. BayesDroid does Bayes classifications on data reaching sink methods. It computes the distance between the data to be sent and the original sensitive data. Given the distance, BayesDroid computes the probability that the leakage is legitimate.

### 2.1.2 Static Taint Analysis

Static taint analysis tools usually decompile application APKs first to obtain the source code of these applications. Based on the code, the tools reconstruct control flow graphs 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. The advantages are that static analysis can make a more thorough analysis of the code in one run. The disadvantages are that the approach runs offline and maybe on a separate machine, which makes the approach difficult to be used by average users. Alternatively, the analysis could be done by an app store. The analysis also tends to be time consuming and may suffer from false positives if a detected flows does not happen in real use. The approach can be imprecise because it needs to abstract from program inputs and approximate run-time objects. Static analysis typically also does not deal with dynamic code loading or native code. FlowDroid [2], EdgeMiner [6], and IFT [9] are three example static taint analysis tools.

## 2.2 Access Interception

We next discuss approaches that intercept an application's access to sensitive data and that block these accesses or modify the data before handing it over to applications.

### 2.2.1 Android Refitting

One way to intercept these accesses is to retrofit Android. AppFence [13] and LP-Guardian [10] are two examples. The

[1] `https://bitbucket.org/Near/privacyguard`
[2] `https://code.google.com/p/sandrop/`

advantage of this approach is no need to modify existing applications, whereas the approach suffers from the disadvantages of having to modify Android, as discussed earlier, and from breaking applications if an application does not expect accesses to be blocked.

### 2.2.2 Application Rewriting

Application rewriting intercepts accesses by modifying applications. It is usually done via rewriting of an application's bytecode. The advantage of this approach is that the underlying platform is unmodified and no flashing or rooting is required. It provides flexible access policies, and there can be different policies for different applications. It also supports different Android versions if the corresponding APIs are not modified. The disadvantages are that applications need to be rewritten before installation, which may be difficult for the average user unless the rewriting is done directly on a device or by an app store. Since policy enforcement becomes part of an application, a malicious application may be able to circumvent this enforcement. Rewriting may violate an application's or app store's EULA. Care needs to be taken when applications are upgraded since rewriting breaks an application's signature. RetroSkeleton [7], Dr. Android and Mr. Hide [14], and AppGuard [5] are three example application rewriting tools.

## 2.3 Heuristics

Fu et al. [11] use two heuristics to provide run-time location access notifications. (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 in the foreground. 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 the function does not change after calling it. Also, applications registering a location listener can obtain location fixes while running in the background.

## 2.4 VPN-based Approaches

ReCon [17] was developed concurrently with PrivacyGuard. ReCon also uses a VPN-based approach to detect information leakage. However, ReCon requires a remote VPN server and therefore has a significantly larger trusted computing base than PrivacyGuard.

Similar to PrivacyGuard, Disconnect[3] uses the `VPNService` class. However, Disconnect addresses a different threat, namely, tracking by advertisement providers. The Disconnect application blocks all communication between applications running on the device and advertisement servers. In particular, the VPN changes the configured DNS server to a DNS server run by the provider of Disconnect, which responds with `127.0.0.1` to all DNS queries for servers run by advertisement providers.

## 2.5 Discussion

All this related work has advantages. However, there are some disadvantages compared to PrivacyGuard. (a) Most work does not consider whether the accessed sensitive data is sent out or in what form it is sent out. This causes false positives and affects usability. (b) Some of the 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) Application rewriting can be difficult for the average user and can cause problems when updating applications.

PrivacyGuard does not have these problems. It works as a regular Android application. Installing and updating PrivacyGuard is easy and does not require root permissions. PrivacyGuard runs continuously on the device and provides real-time protection. Alerts are generated only when sensitive data is sent out. Sensitive data that has been processed and is no longer sensitive is ignored.

# 3. DESIGN OPTIONS

We are concerned with information leakage over a network. Intercepting and inspecting the network traffic of applications could therefore allow us to detect this leakage.

## 3.1 Goals

We argue that a solution for intercepting network traffic should at least have the following properties:

- **Functionality:** The solution should intercept all network traffic of an application, including traffic encrypted with TLS.
- **Usability:** The solution 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 Android version 4.0 or later and to any Android device.
- **Acceptable overhead:** The solution monitors all network traffic and runs continuously on a device. Therefore, it should not use much battery power. Since it provides real-time analysis of the network traffic, it should have limited effect on network performance.
- **Extensibility:** The solution should be extensible. Developers should be able take advantage of the access to network traffic provided by the solution to develop their own network filtering plugins. Privacy researchers should be able to use the solution to prototype their algorithms to detect information leakage or to alert users of this leakage.

We require an approach that gives us access to all network traffic, similar to *tcpdump* or *wireshark*. Since Android is based on Linux, porting *tcpdump* is feasible (in fact, there is a *tcpdump* for Android[4]). However, running *tcpdump* requires root permissions, which is unacceptable to us.

There are two alternative ways ways to gain access to network traffic: using a proxy or using a VPN.

A proxy is a computer system or an application that acts as an intermediary for requests from clients seeking resources from other servers. Proxies can have several types, such as a Web proxy or a SOCKS proxy. A web proxy cannot fulfill our requirements because it can forward only HTTP packets. A SOCKS proxy supports both TCP and UDP. Due to restrictions of Android, no application can configure a

[3] `https://disconnect.me/`

[4] `http://www.androidtcpdump.com`

proxy automatically without root permissions. It is necessary to manually configure the proxy for every WiFi connection. This manual configuration can be a problem for average users.

We choose to use a VPN, which refers to a 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 tunnel through the use of dedicated connections. VPNs work on the IP layer. Android has built-in support for making applications send their traffic through a VPN tunnel. Also, an Android application can programmatically set up a VPN tunnel that will be used by all applications, and the only required user interaction is tapping OK on a prompt once.

## 3.2 Remote VPN Server

The easiest way is by setting up a remote VPN server and having the VPN application route all traffic through this server. The server analyzes the traffic and reports all information leakage to the users. A remote VPN server is easy to set up and use, but the approach has some problems:

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

To avoid these problems, we propose the second option.

## 3.3 Local VPN Server

A VPN server that runs locally on the device 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 to us. In addition, we observe that we do not need a fully-featured VPN server. We need access to all network traffic, but do not need any other features typically provided by a VPN, such as encryption.

## 3.4 Fake VPN Server

We can use the `VPNService` class provided by the Android SDK starting with 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 always start with IP headers. The application then completes a VPN connection by processing and exchanging packets with the remote VPN server over a tunnel.

In our case, there is no remote VPN server. The `VPNService` only pretends that it has connected to such a server.

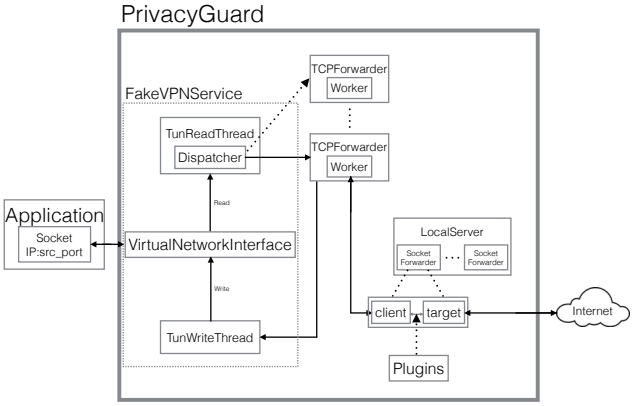



Figure 1: Architecture of PrivacyGuard.

PrivacyGuard obtains network traffic from the virtual network interface provided by `VPNService` and retransmits it after the analysis. The details can be found in section 4.

## 3.5 Challenges

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

- 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 analyze traffic that is transmitted by applications in encrypted form with the TLS protocol.

# 4. IMPLEMENTATION

Figure 1 presents the five main components of PrivacyGuard: `FakeVPNService`, `TCPForwarder` (or `UDPForwarder`), `LocalServer`, `SocketForwarder`, and `Plugins`.

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

## 4.1 Basic Work Flow

The following is the basic work flow of PrivacyGuard:

1. An application sends a request to a server. The request is then 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 intended server in the Internet (from now on called "real server") and also retransmits the response from the real server to the `TCPForwarder`. In this step, all installed `Plugin`s are invoked to filter outgoing and incoming data.

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 the following sections.

## 4.2 The `FakeVPNService` Class

The class `VPNService`[5] has been part of the Android SDK since 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.

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

The `TunReadThread` thread reads all requests from the virtual network interface. Because many applications and services can run on the device, the amount of requests can be large. There would be heavy performance overhead due to the high network I/O cost if the `TunReadThread` thread handled the transmission to other components as well. To avoid this cost, this thread only adds these requests to a queue.

The `Dispatcher` thread is created by the `TunReadThread` thread. The `Dispatcher` thread keeps reading requests, which are IP datagrams since VPN works on the IP layer, from the queue mentioned above. The `Dispatcher` thread parses these IP datagrams and 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` thread 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.

The `TunWriteThread` thread 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.

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

## 4.3 The `TCPForwarder` Class

### 4.3.1 TCP Connection States

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 source IP address, which is the same for all applications, and the source port number, the relationship is based on the source port number.

`TCPForwarder` implements the TCP state machine. The following explains the implementation through a typical TCP connection life cycle:

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.

The TCP connection states and related information are stored in `TCPConnectionInfo`s. The `TCPConnectionInfo` class implements all necessary methods to read and update the states.

Because TCP is a stream delivery service, one request of a protocol using TCP (e.g., HTTP) may be divided into multiple 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.

### 4.3.2 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 to real servers (sent by the application) from the `FakeVPNService` and responses from the real servers.

Each `TCPForwarder` connects to the `LocalServer` like it would connect to a normal server by creating a socket and connecting it to the `LocalServer`. The socket binds to `127.0.0.1` and the source port in the TCP packets, that is, the port number used by the corresponding application.

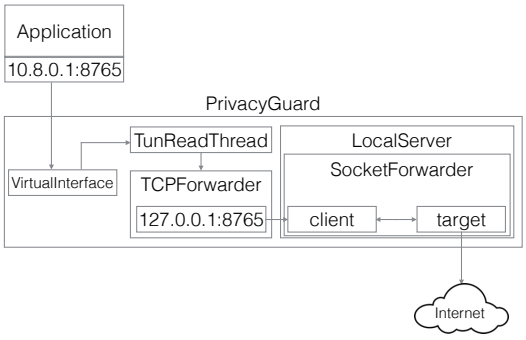



Figure 2: Configuration of the sockets used in PrivacyGuard.

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 2 shows this configuration. The reason we use the same port number is described in section 4.4.

The two directions for the transmissions between a `TCPForwarder` and the `LocalServer` are treated differently.

For transmissions from a `TCPForwarder` to the `LocalServer`, the `TCPForwarder` retrieves the TCP data from IP datagrams and then passes the data through the socket.

For transmissions from the `LocalServer` to a `TCPForwarder`, we observe that since `TCPForwarder`s 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 4.2). We need to create an IP datagram by using the data and the status we maintain in `TCPConnectionInfo`s (see section 4.3.1) and `TCPForwarder`s. TCP checksum recalculation is also necessary.

## 4.4 The `LocalServer` Class

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

When a `TCPForwarder` connects to the `LocalServer`, the `LocalServer` creates two sockets: `target` and `client`. Then the `LocalServer` creates a `SocketForwarder` (described in section 4.5) using these two sockets (see figure 2). The `target` socket connects to the real server; its IP address and port number are determined as follows:

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 4.3.2). Therefore, we can use the 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`. With these files, we can also determine the UID of the application that issued this connection. The application name can be obtained with the UID as well.

## 4.5 The `SocketForwarder` Class

### 4.5.1 Basic Forwarding

`SocketForwarder`s match one to one with TCP connections. Each `SocketForwarder` contains two sockets, the `client` socket and the `target` socket, and a pair of threads passing data between them.

The outgoing thread receives data from the `TCPForwarder` through the `client` socket. It filters the data (see section 4.6) and forwards the data to the real server through the `target` socket. The current filtering approach is based on string matching, which can take considerable time for long messages. Therefore, 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 allows modifications to data before sending the data.

2. Asynchronous Filtering: adding data to a queue after reading it from the `client` socket and sending the data to the `target` socket right away. Another thread retrieves data from the queue and does the filtering. With this option, the filtering causes less network overhead, but we can detect leaks only retroactively.

The incoming thread receives data from the real server through the `target` socket. It filters the data and forwards the data to the `TCPForwarder` through the `client` socket.

### 4.5.2 Handling TLS Connections

As mentioned in section 3.5, the network traffic analysis requires plain text. However, TLS connections encrypt network traffic. If a client wants to establish an TLS connection with a server, the client and the server execute a TLS 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. (There are also other types of handshakes, see section 4.7.) Obtaining the pre-master secret and the plain text of the following messages requires the private key for the public key used for encrypting the pre-master-secret. However, this private key is not available to PrivacyGuard.

To reveal the data, we deploy a man-in-the-middle proxy instead, where PrivacyGuard pretends to be the server in view of the client and the client in view of the server. When a client wants to access the server, PrivacyGuard 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 PrivacyGuard since the public/private key pair belongs to PrivacyGuard. For every message from the client, PrivacyGuard reencrypts it and transmits it to the server.

When PrivacyGuard launches the first time, it creates a root CA certificate containing its public key and installs the certificate on the device. Every certificate signed by this root CA certificate is then trusted by applications.

When an application wants to establish a TLS 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 a TLS connection with the destination IP address.

```
public interface IPlugin {
  // filter on requests and responses.
  public String handleRequest(String request);
  public String handleResponse(String response);

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

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

Figure 3: `IPlugin` Plugin Interface.

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 TLS connections. These certificates are signed by the root CA certificate. With fake certificates, the `LocalServer` can finish the TLS handshake with applications. Both the `client` and `target` sockets are also replaced with `SSLSockets`. Although the traffic between applications and the `client` socket and between the `target` socket and the real server is encrypted, we can now read data from the `client` and `target` sockets in plain text. Our source code for certificate generation and hostname lookup is from SandroProxy[6].

The usage of TLS interception has been controversial [20]. However, there are many benevolent use cases for TLS interception [15], and we argue that PrivacyGuard is one of them. First, we are transparent about PrivacyGuard and its purpose, and we let the phone owner himself/herself decide whether to install it. Second, PrivacyGuard uses the default TLS routines in Java for certificate checking and does not replace them with weakened ones, as is still done by many applications [16]. In turn, this implies that for many applications PrivacyGuard actually increases their security. Because PrivacyGuard is open-source, anyone could add add more sophisticated certificate checking routines (e.g., certificate pinning) to PrivacyGuard. Third, PrivacyGuard generates a fresh public/private key pair for the root CA on each device instead of re-using the same pair across all devices.

## 4.6 Customized Plugins

Plugins filter outgoing and incoming network traffic. All plugins must implement the `IPlugin` interface shown in figure 3. It is possible to filter incoming responses but our sample plugins are targeted at detecting information leakage and filter only outgoing requests. The corresponding methods of all installed plugins will be called to filter traffic. With plugins, developers can easily set up their own analysis tools.

We have implemented three plugins (see section 5.4).

- LocationDetection: detects phone's location.
- PhoneStateDetection: detects phone's IMEI, IMSI, and AndroidID.
- ContactDetection: detects email address and phone number of phone owner.

[6] `https://code.google.com/p/sandro/`

## 4.7 Limitations

The current implementation of PrivacyGuard has some limitations:

- If an application itself (i.e., not TLS) encrypts sensitive information before its transmission, PrivacyGuard will fail to detect this leakage since our plugins filter with string matching. In 53 evaluated applications (see section 5), we observe three applications where we suspect that the application encrypts transmitted data.
- Although our man-in-the-middle proxy works well in most situations, it cannot address certificate pinning. If an application uses certificate pinning, the application will store the certificates it trusts locally. For any certificate this application receives while establishing a TLS 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 certificate pinning unless the proxy modifies the certificates saved by the application. In our evaluation, we observe that only the Twitter application uses certificate pinning.
- Currently, our man-in-the-middle proxy supports only RSA for the key exchange executed during the TLS handshake because we use a library from SandroProxy, which can generate only RSA key pairs. However, additional key-exchange methods can be added to our open-source implementation.
- PrivacyGuard cannot distinguish legitimate leakage of sensitive data (i.e., leakage required to provide an application's core functionality) from illegitimate leakage. This limitation is also a common problem of other work. One possible solution is using Bayes classification [19]. 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 [10], we can also use the host or IP address information to distinguish between legitimate and illegitimate sharing.
- TaintDroid can track data from gyroscopes, cameras, and microphones. PrivacyGuard cannot detect this kind of data since it would be too expensive to filter.

# 5. EFFECTIVENESS

In this section, we examine one possible usage scenario of PrivacyGuard. Namely, we evaluate whether PrivacyGuard can effectively detect information leakage by applications. We compare our results with TaintDroid since TaintDroid is a popular state-of-the-art taint-based privacy detection tool.

Whereas we have successfully used PrivacyGuard for modifying sensitive information before transmitting it to the real server, we leave a thorough evaluation of this aspect for future work.

## 5.1 Setup

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

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

Initially, our goal was to run all experiments in the emulators because we have only one device and it is easier to automate experiments in the emulators. However, some components of the Google Play service required by some applications are missing in the emulators. We tried to set up these components ourselves. 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 applications from Goo.im[7] and pushed these to the emulators. The framework is not complete, but some applications requiring the Google Play service can now run in the emulators.

Due to remaining difficulties, such as the still incomplete Google Play service and other missing features in the emulators, such as the network location provider, we also use a real device to evaluate applications that do not run in the emulators. Since we have only one device, we run both TaintDroid and PrivacyGuard on it. TaintDroid does not track information flow through third-party native libraries so by default TaintDroid refuses to execute applications that use third-party native libraries. PrivacyGuard is oblivious to this aspect since it analyzes only network traffic. We therefore modified TaintDroid to allow the execution of applications with third-party native libraries. This modification is also necessary to run PrivacyGuard itself because PrivacyGuard requires a native library from SandroProxy for hostname lookup. Our modification does not reduce the number of applications for which TaintDroid can detect an information leakage. At best, it increases TaintDroid's number of detections in cases where an application with a third-party native library occurs a leakage during the execution of code that is not using such a library.

## 5.2 Methodology

Our effectiveness evaluation consists of two phases. In the training phase, we manually analyze the network traffic of some applications for information leakage and derive filters to be used by PrivacyGuard. We use the output of TaintDroid as the ground truth. In the testing phase, we use the filters to automatically analyze network traffic and compare our results with TaintDroid's results.

The 53 applications that we use for training and testing have been used in other papers [19][10]. We use 13 out of 15 applications from table 3 of Tripp and Rubin [19] since the other two are not available from Google Play anymore. We also acquired the list of applications used by Fawaz et al. [10]. Several applications in the list are not available anymore. For an evaluated application, we do the following:

1. We first uninstall the application if it is already installed to clear caches and then install it.
2. Before using the application, we set up the GPS location provider in the emulators and launch a helper application that reads location values from the provider such that the `getLastKnownLocation()` method will return a valid location to the evaluated application later. Running this helper application is necessary because some applications obtain the location only by calling this method, instead of registering a location update listener.
3. We run the application in the two emulators in parallel and manually explore its UI and activities. We make sure to execute each input operation in both emulators at the same time. If the application does not run in the emulators, we explore the application on the device that runs both TaintDroid and PrivacyGuard. We intended to use PUMA [12] for automatic exploration. However, the exploration approach that PUMA has included is not able to recognize all UI components, and we still have to manually type the username and password required by some applications, such as Facebook.

[7] `https://goo.im/`

## 5.3 Training Phase

We use ten applications from our list of applications for training. 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 1. The checkmark means the corresponding information is detected in the network traffic of that application.

The sensitive data that we try to detect and the corresponding filters that we derived are shown in table 2. For the location data (i.e., latitude and longitude), the LocationDetection plugin itself needs to obtain the phone's location first. However, invoking `getLastKnownLocation()` for each message takes too much time. Instead, the plugin registers a `LocationListener` for every available location provider. (Location providers are a concept of Android. Developers can obtain location data from available location providers.) This way the plugin retrieves the current location from all location providers and keeps two decimal points of the received floating point values as filters. For the device identifiers, the PhoneStateDetection plugin retrieves these values using the corresponding Android API calls and then looks for these values in the network traffic. In addition, the plugin computes the SHA-1 and MD5 hashes of these values and also uses these hashes as filters. Some applications send these hashes instead of the original values. For the contact information, the ContactDetection plugin retrieves the phone number and the email address of the phone owner and looks for these values in the outgoing network traffic using a regular expression that discards unnecessary white/filler space. Our 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 fil-

Table 1: List of training applications.

| 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 | | ✓ | | | ✓ | |

Table 2: Filters used by PrivacyGuard.

| Data | Source | Filters |
|---|---|---|
| Location | all available location providers | keep two decimal points |
| Dev.ID | IMEI, IMSI, AndroidID | plain text, SHA-1, MD5 |
| Contact | phone number, email address | regular expression |

Table 3: Summary of results.

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

ters for these packets. Some of these detected leaks are links to pictures. We suspect that these leaks are false positives.

PrivacyGuard fails for the Twitter application since this application uses certificate pinning and PrivacyGuard cannot intercept its plain text traffic (see section 4.7).

## 5.4 Testing Phase

For PrivacyGuard, we use the filters from the training phase to analyze network traffic in the testing phase. For TaintDroid, we run it and record its notifications. Table 3 shows how many applications have information leakage detected by TaintDroid and PrivacyGuard. Overall, PrivacyGuard detects more leakage incidents than TaintDroid.

The detailed, per-application results are given in table 4. 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, PrivacyGuard can detect more leaks than TaintDroid. However, there are some applications for which PrivacyGuard is unable to detect information leakage, such as *com.starbucks.mobilecard* or *com.dictionary.com*. These applications use the Google Maps API for location-based services. Inspecting the network traffic indicates that this API obfuscates the location data. The obfuscation makes it almost impossible for PrivacyGuard to detect the leakage. This limitation is also mentioned in section 4.7.

We also run PrivacyGuard and TaintDroid on some of the most popular advertisement libraries. For each library, we develop a dummy application to wrap it. As shown in table 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 Android code not instrumented by TaintDroid to round up floating point numbers.

# 6. PERFORMANCE EVALUATION

For the performance evaluation of PrivacyGuard, we focus on its network performance and battery consumption. We deploy two Android applications on the Nexus 4 device mentioned in Section 5.1.

- UpDownLoader, an Android application developed by ourselves with 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 battery consumption

  The application cooperates with a dummy server running on a desktop.

- SpeedTest.net[8], an Android application for testing network speed and ping delay.

## 6.1 Network Performance

Section 4.6 explains that filtering is done by string matching. String matching may take a lot of time, especially if the network message is large. Many memory allocations as well as string operations, such as concatenation or copy, are executed. These operations cost time and thus may cause network delay since messages are sent out only after the filtering in the synchronous configuration (see section 4.5.1). 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 both latency and throughput.

Four scenarios are tested: no PrivacyGuard, PrivacyGuard without filtering for information leakage, PrivacyGuard with synchronous filtering, and PrivacyGuard with asynchronous filtering. Both types of filtering use three filters; one for each of location, contact information, and device ID. For each of the first two scenarios, separate experiments are executed with different settings described below. Since our sample plugins filter only outgoing traffic, we test the two filtering scenarios only with uploading files.

- Download and upload one 1 Mbytes file with UpDownLoader.

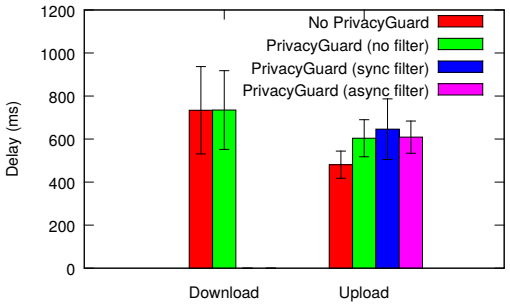
[8] `https://play.google.com/store/apps/details?id=org.zwanoo.android.speedtest&hl=en`

Table 4: List of test applications.

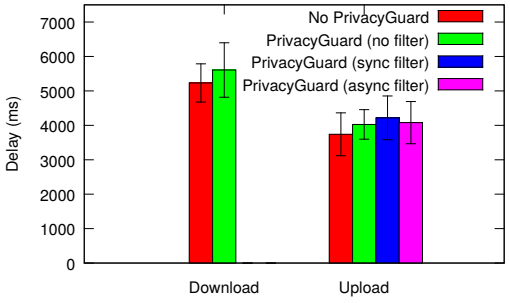
| App | Category | TaintDroid | | | PrivacyGuard | | |
|---|---|---|---|---|---|---|---|
| | | Dev.ID | Location | Contact | Dev.ID | Location | Contact |
| com.google.android.apps.maps | Travel&Local | | | | | ✓ | |
| com.facebook.katana | Social | | ✓ | | | ✓ | |
| com.android.chrome | Communication | | ✓ | | | ✓ | |
| 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 5: List of advertisement libraries.

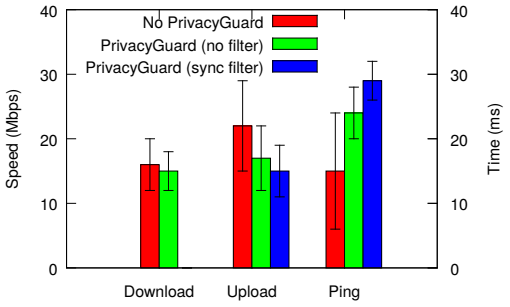
| App | Category | TaintDroid | | | PrivacyGuard | | |
|---|---|---|---|---|---|---|---|
| | | Dev.ID | Location | Contact | Dev.ID | Location | Contact |
| admob | Ad Library | | | | | | |
| amazon | Ad Library | | | | | ✓ | |
| airpush | Ad Library | | ✓ | | | ✓ | |
| inmobi | Ad Library | | ✓ | |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




(a) 1 Mbytes file experiments.




(b) 10 Mbytes file experiments.




(c) SpeedTest.net experiments.

Figure 4: Results of network performance evaluation. Note the different units and scales for the y-axes.

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

No other applications causing network traffic are running concurrently with UpDownLoader or SpeedTest.net.

Because we use the WiFi network to run these experiments, we run the same experiment ten 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.

The experiment results are shown in figure 4. All figures show the average values and the standard deviations.

For the downloading part of figures 4a, 4b, and 4c, the downloading delay is similar between different scenarios. This observation is as expected because as described in section 4.6, our sample plugins do not filter the incoming network traffic and thus do not introduce much overhead.

For the uploading part, PrivacyGuard with synchronous filtering is 34.3% (165 ms) and 12.8% (479 ms) slower than no PrivacyGuard in the 1 Mbytes and 10 Mbytes experiments, respectively. PrivacyGuard with asynchronous filtering is 26.6% (128 ms) and 9.1% (339 ms) slower than no PrivacyGuard in the 1 Mbytes and 10 Mbytes experiments, respectively. For PrivacyGuard without filtering, it is 25.6% (123 ms) and 7.7% (287 ms) slower than no PrivacyGuard running in the 1 Mbytes and 10 Mbytes experiments. In conclusion, filtering does not introduce much overhead. Furthermore, the relative overhead in uploading gets lower as the size of the file increases.

For the experiments with SpeedTest.net, the downloading results are similar for the two settings. For the uploading speed, PrivacyGuard without filtering is 22.2% slower than no PrivacyGuard and PrivacyGuard with filtering is 30.2% slower. While doing the experiments, we observe that the SpeedTest.net application reaches high upload speeds at the beginning of an experiment with PrivacyGuard running, but there are 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. The SpeedTest.net application opens multiple TCP connections in parallel for uploading or downloading. EPIPE errors shut down some of these connections and slow down the speed. Our analysis of the TCP packets to see if PrivacyGuard sends any invalid packets that could cause this error did not return any such packets. Since PrivacyGuard achieves similar uploading speeds as the scenario without PrivacyGuard in some experiment runs, we are confident that PrivacyGuard can achieve higher average speeds once we get rid of these errors. We note that when replacing PrivacyGuard with tPacketCapture[9], a closed-source application that uses `VPNService` for packet capture, we also observe slow uploading speeds. For the ping delay, PrivacyGuard introduces about 59.3% overhead without filtering and 92.0% overhead with synchronous filtering. Although the percentage is high, the absolute increase of the ping delay remains acceptable. PrivacyGuard's ping delay is 26 ms on average, only 11 ms longer than no PrivacyGuard.

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 packets are small.

## 6.2 Battery Consumption

To measure the battery consumption of PrivacyGuard, we consider three scenarios: no PrivacyGuard running, PrivacyGuard without filtering, and PrivacyGuard with synchronous filtering. For each scenario, we use UpDownLoader to run separate experiments for downloading and uploading. For each experiment, we follow these steps:

1. Fully charge the device.
2. Connect to the dummy server running on a desktop.
3. Start downloading or uploading a 1 Mbytes file every 10 seconds for 60 minutes.

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

Table 6: Decrease in battery level.

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

4. Turn off the screen.
5. The device will vibrate and record the decrease in battery level, retrieved using the Android API, to a file when the task finishes.

The results are shown in table 6. PrivacyGuard introduces almost no additional battery consumption even with a high frequency of uploading and downloading. This result is expected since we observe no significant increase on the CPU load by PrivacyGuard.

# 7. CONCLUSION AND FUTURE WORK

We propose and implement a new approach, PrivacyGuard, to filter network traffic on Android. PrivacyGuard does not require root permissions and is portable to all devices with Android versions 4.0 or later. 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 TLS traffic. PrivacyGuard can be used to effectively detect information leakage. It introduces acceptable overhead in both network performance and battery consumption.

PrivacyGuard provides a platform for developers to develop their own filtering 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. Future work should consider adding a user-friendly interface to PrivacyGuard to help users configure their plugins as well as avoid distraction.

## Acknowledgments

We thank Martin Karsten, Ian McKillop, and the anonymous reviewers for their helpful comments. This work is supported by a Google Focused Research Award, the Natural Sciences and Engineering Research Council of Canada, and the Ontario Research Fund.

# 8. REFERENCES

[1] H. Almuhimedi, F. Schaub, N. Sadeh, I. Adjerid, A. Acquisti, J. Gluck, L. Cranor, and Y. Agarwal. Your location has been shared 5,398 times! a field study on mobile app privacy nudging. In *CHI*, 2015.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, 2014.

[3] K. W. Y. Au, Y. F. Zhou, Z. Huang, P. Gill, and D. Lie. Short paper: a look at smartphone permission models. In *SPSM*, 2011.

[4] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the Android permission specification. In *CCS*, 2012.

[5] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. 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] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. Edgeminer: automatically detecting implicit control flow transitions through the Android framework. In *NDSS*, 2015.

[7] B. Davis and H. Chen. Retroskeleton: retrofitting Android apps. In *MobiSys*, 2013.

[8] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[9] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu. Collaborative verification of information flow for a high-assurance app store. In *CCS*, 2014.

[10] K. Fawaz and K. G. Shin. Location privacy protection for smartphone users. In *CCS*, 2014.

[11] H. Fu, Y. Yang, N. Shingte, J. Lindqvist, and M. Gruteser. A field study of run-time location access disclosures on Android smartphones. In *USEC*, 2014.

[12] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *MobiSys*, 2014.

[13] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In *CCS*, 2011.

[14] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: fine-grained permissions in Android applications. In *SPSM*, 2012.

[15] M. O'Neill, S. Ruoti, K. E. Seamons, and D. Zappala. TLS proxies: friend or foe? *CoRR*, abs/1407.7146, 2014.

[16] L. Onwuzurike and E. De Cristofaro. Danger is my middle name: experimenting with SSL vulnerabilities in Android apps. In *WiSec*, 2015.

[17] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes. ReCon: Revealing and controlling privacy leaks in mobile network traffic. *CoRR*, abs/1507.00255, July 2015.

[18] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices. In *SECRYPT*, 2013.

[19] O. Tripp and J. Rubin. A Bayesian approach to privacy enforcement in smartphones. In *USENIX Security*, 2014.

[20] Wikipedia. Superfish. `https://en.wikipedia.org/wiki/Superfish`. Accessed July 2015.