# Exploiting Data-Usage Statistics for Website Fingerprinting Attacks on Android

Raphael Spreitzer, Simone Griesmayr, Thomas Korak, and Stefan Mangard
Graz University of Technology, IAIK, Austria
raphael.spreitzer@iaik.tugraz.at

## ABSTRACT

The browsing behavior of a user allows to infer personal details, such as health status, political interests, sexual orientation, etc. In order to protect this sensitive information and to cope with possible privacy threats, defense mechanisms like SSH tunnels and anonymity networks (e.g., Tor) have been established. A known shortcoming of these defenses is that website fingerprinting attacks allow to infer a user's browsing behavior based on traffic analysis techniques. However, website fingerprinting typically assumes access to the client's network or to a router near the client, which restricts the applicability of these attacks.

In this work, we show that this rather strong assumption is not required for website fingerprinting attacks. Our client-side attack overcomes several limitations and assumptions of network-based fingerprinting attacks, e.g., network conditions and traffic noise, disabled browser caches, expensive training phases, etc. Thereby, we eliminate assumptions used for academic purposes and present a practical attack that can be implemented easily and deployed on a large scale. Eventually, we show that an unprivileged application can infer the browsing behavior by exploiting the unprotected access to the Android data-usage statistics. More specifically, we are able to infer 97% of 2 500 page visits out of a set of 500 monitored pages correctly. Even if the traffic is routed through Tor by using the Orbot proxy in combination with the Orweb browser, we can infer 95% of 500 page visits out of a set of 100 monitored pages correctly. Thus, the `READ_HISTORY_BOOKMARKS` permission, which is supposed to protect the browsing behavior, does not provide protection.

## Keywords

Mobile security; side-channel attack; website fingerprinting; data-usage statistics; mobile malware

## 1. INTRODUCTION

Mobile devices, like smartphones and tablet computers, are widely employed and represent an integral part of our everyday life. Due to this tight integration, these devices store and process sensitive information. In order to protect this data as well as the users' privacy, appropriate mechanisms must be implemented. For instance, the Android operating system relies on two fundamental security concepts, namely the concept of sandboxed applications and a permission system. Sandboxing is ensured by the underlying Linux kernel by assigning each application a unique user ID (UID). This means that an application's resources can only be accessed by this application itself, whereas applications running in parallel on the same device do not gain direct access to other applications. The permission system ensures that applications must explicitly request specific permissions for dedicated resources, for example, access to the GPS sensor or the Camera, which might harm the users' privacy.

Although Android relies on the two concepts of sandboxing and permissions, applications running in parallel on the same device are still able to gain information about other applications by exploiting shared resources. Examples are the list of installed applications, the memory footprint of applications, the data-usage statistics, and also the speaker status (speaker on/off). Even though this information seems to be innocuous, sensitive information can be inferred as has been demonstrated by Jana and Shmatikov [22] and Zhou et al. [44]. Studies like these do not exploit specific vulnerabilities of applications but investigate and demonstrate weaknesses of fundamental security concepts on mobile devices. In order to advance the field of mobile security and to protect the user's privacy, a thorough understanding regarding the limitations of fundamental security concepts is required.

In this work, we study the information leakage of the publicly available data-usage statistics on Android. More specifically, Android-based smartphones track the amount of incoming/outgoing network traffic on a per-application basis. This information is used by data-usage monitoring applications to inform users about the traffic consumption. However, while this feature might be helpful to stick to one's data plan and to identify excessive data consumptions of applications, we show that this seemingly innocuous information allows to infer a user's visited websites. Thereby, we demonstrate that the `READ_HISTORY_BOOKMARKS` permission, which is intended to protect this sensitive information, is actually useless as any application without any permission at all is able to infer visited websites rather accurately.

The exploitation of observed "traffic information" to infer visited websites is known as website fingerprinting [21, 37]. Thereby, an attacker aims to match observed "traffic information" to a previously established mapping of websites and

*WiSec'16, July 18–22, 2016, Darmstadt, Germany.*

ISBN 978-1-4503-4270-4/16/07...$15.00

DOI: http://dx.doi.org/10.1145/2939918.2939922

their corresponding "traffic information". Most of the investigations in this area of research consider an attacker who sniffs the traffic information "on the wire". This means that the attacker needs to be located on the client's network or on the ISP's router near the client. However, as Android allows an attacker to capture the required data directly on the smartphone without any permission, we show that an attacker is not required to be located somewhere on the victim's network. Hence, the rather strong assumption of a network-based attacker is not required for website fingerprinting attacks. Furthermore, our attack is invariant to traffic noise of other applications—one of the major drawbacks of network-based attacks—as Android captures these statistics on a per-application basis. Compared to existing website fingerprinting attacks, we significantly reduce the computational complexity of classifying websites as we do not require a dedicated training phase, which sometimes requires several hundred CPU hours [38, 39]. Instead, we rely on a simple yet efficient classifier.

Based on our observations, we developed a proof-of-concept application that captures the data-usage statistics of the browser application. With the acquired information in a closed-world setting of 500 monitored websites of interest, we are able to classify 97% of 2 500 visits to these pages correctly. The fact that not even Tor on Android (the *Orbot*[1] proxy in combination with the *Orweb*[2] browser) is able to protect the user's page visits clearly demonstrates the rather delicate issue of this fundamental design flaw.

## 1.1 Further Security Implications

We stress that the presented attack can be combined with related studies to obtain even more sophisticated attack scenarios. For instance, sensors employed in mobile devices—including the accelerometer, the gyroscope, and also the ambient-light sensor—have been shown to be exploitable in order to infer the user's keyboard inputs without any permission (cf. [1, 3, 31, 34, 36, 41]). In combination with such sensor-based keyloggers, our attack would allow an adversary to determine when a user visits a specific website and to gather login credentials for specific websites. Such an attack does not only endanger the users' privacy but also allows for large-scale identity theft attacks. Thus, OS developers need to deal with this problem in order to prevent users from such severe privacy threats and identity thefts.

## 1.2 Contribution

The contributions of this work are as follows. First, we investigate the information leakage through the data-usage statistics published by the Android OS. Based on this investigation, we provide an adversary model for a realistic attack scenario that allows for large-scale attacks against the browsing behavior of smartphone users. Furthermore, we discuss how to capture the required information and we show how to infer the browsing behavior with a high accuracy, including a setting where traffic is routed through the anonymity network Tor. We compare our results with existing fingerprinting attacks and show that our attack (1) outperforms these attacks in terms of accuracy, (2) can be deployed significantly easier than existing attacks, and (3) is more efficient in terms of computational complexity. Hence, our attack can be easily deployed on a large scale.

## 1.3 Outline

The remainder of this paper is organized as follows. In Section 2, we introduce website fingerprinting and related work. In Section 3, we discuss the feature of data-usage statistics captured by the Android OS and how this information relates to the actual transmitted TCP packets. Based on these observations, we outline the adversary model and possible attack scenarios in Section 4 and we discuss our chosen attack approach in Section 5. We extensively evaluate the results in Section 6 and discuss possible countermeasures in Section 7. Finally, we conclude this work in Section 8.

# 2. BACKGROUND AND RELATED WORK

Website fingerprinting can be considered as a supervised machine-learning problem, namely a classification problem. The idea is to capture the "traffic signature" for specific websites—which are known to the attacker—during a training phase. The "traffic signature" consists of specifically chosen features like unique packet lengths, packet length frequencies, packet ordering, inter-packet timings, etc. In order to capture this information, the attacker loads different websites and observes the resulting "traffic signature", which is usually done somewhere on the network. During the attack phase, an observed "traffic signature" can be classified according to the previously trained classifier.

Most related work in the context of website fingerprinting attacks operate in the closed-world setting. In contrast to the open-world setting, the closed-world setting assumes that the victim only visits a specific set of monitored websites. Furthermore, most fingerprinting attacks assume a passive attacker, although studies considering an active attacker also exist [19]. In contrast to passive attackers, active attackers can influence the transmitted packets, e.g., by delaying specific packets. Thus, active attacks rely on stronger assumptions of the attacker's capabilities.

Subsequently, we summarize related work according to the exploited information and how this information is gathered. We start with attacks that require access to the victim's network trace or to the ISP's router near the victim. Afterwards, we continue with remote adversaries and finally we discuss the exploitation of shared resources on the victim's device, which is the category of attacks our work belongs to.

## 2.1 On the Wire

Back in 2002, Hintz [21] mentioned that encrypted traffic does not prevent an adversary from inferring a user's visited website. In fact, the simple observation of the amount of transferred data—which is not protected by means of SSL/TLS (cf. [30])—allows an adversary to infer the visited website. Similarly, Sun et al. [37] mentioned that the observation of the total number of objects and their corresponding lengths can be used to identify websites, even if the content itself is encrypted.

While early studies [21, 37] exploited the actual size of web objects, a more recent study [27] focused on the exploitation of individual packet sizes. Such fingerprinting attacks have been demonstrated to work against the anonymity network Tor [6, 35, 39] and also against WEP/WPA encrypted communication as well as IPsec and SSH tunnels [2, 28]. Recently, also evaluations of different attacks and countermeasures under different assumptions have been done [5, 24].

Instead of inferring specific websites, Chen et al. [7] extracted illnesses and medications by observing the sequence

[1] https://guardianproject.info/apps/orbot

[2] https://guardianproject.info/apps/orweb

of packet sizes. Similarly, Conti et al. [8] inferred a user's interaction with specific Android applications, e.g., *Gmail*, *Twitter*, and *Facebook*, based on the transmitted packets. In order to eavesdrop on the transmitted packets, they routed the network traffic through their specifically prepared server.

## 2.2 Remote

Timing attacks on the browser cache [14, 26] can be used to infer whether or not a user visited a specific website before. More specifically, by measuring the loading time of a specific resource, an attacker can determine whether it was served from the browser's cache or not. Similarly, CSS styles of visited URLs can be used to determine the browsing behavior [23]. Gong et al. [15, 16] even demonstrated that fingerprinting can be done remotely when given the victim's IP address. Therefore, the attacker sends ping requests to the user's router and computes the round-trip time, which correlates with the victim's HTTP traffic. Another remote exploitation has been demonstrated by Oren et al. [33], who showed that JavaScript timers can be used to distinguish between memory accesses to the CPU cache and the main memory, which allows for so-called cache attacks via JavaScript. Based on these precise JavaScript timers, they were able infer page visits to a set of 8 websites. Similarly, Gruss et al. [17] exploited so-called page-deduplication attacks to infer page visits to a set of 10 websites.

## 2.3 Local

In 2009, Zhang and Wang [42] exploited the `/proc` file system to infer inter-keystroke timings and argued that the privacy risks of the `/proc` file system need to be investigated further. This has been done, for instance, by Jana and Shmatikov [22] who demonstrated the possibility to fingerprint websites based on the browser's memory footprint, which is available via the `/proc` file system. In addition, Zhou et al. [44] demonstrated that the data-usage statistics of Android applications can be used to infer the user's activities within three Android applications, namely *Twitter*, *WebMD*, and *Yahoo! Finance*. Later, Zhang et al. [43] exploited the data-usage statistics of an Android-based Wi-Fi camera to determine when the user's home is empty.

Even though Zhou et al. [44] and Zhang et al. [43] started the investigation of the information leakage through the data-usage statistics on Android devices for specific applications, a detailed study regarding the applicability of this information leakage to infer websites has not been done yet. Compared to the work of Jana and Shmatikov [22] who exploit the memory footprint of the browser application for website fingerprinting attacks, we demonstrate a significantly more accurate attack by exploiting the data-usage statistics.

# 3. ANDROID DATA-USAGE STATISTICS

Android keeps track of the data usage in order to allow users to stick to their data plan. This accounting information is available through the public API as well as through the `/proc` file system. More specifically, the `TrafficStats` API as well as `/proc/uid_stat/[UID]/{tcp_rcv|tcp_snd}` provide detailed information about the network traffic statistics on a per-UID basis. Since every Android application is assigned a unique UID, these traffic statistics are gathered on a per-application level. In order to observe the data-usage statistics of an application, e.g., the browser, the corresponding UID is required. This information can be retrieved via the `ActivityManager` API for all running processes.

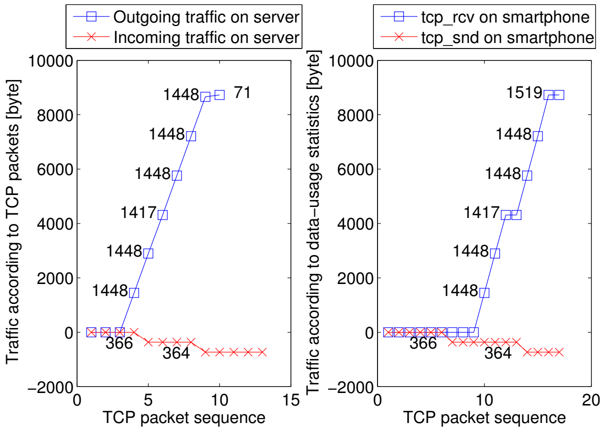

**Figure 1: TCP packet lengths according to `tcpdump` and data-usage statistics on the smartphone.**

Subsequently, we investigate the information leakage of the data-usage statistics in more detail. We study the information leakage for browser applications considering a standard setting and in case the traffic is routed through the Tor network. Furthermore, we also investigate the information leakage depending on the network connection. Finally, we discuss our observations regarding the API support for the data-usage statistics and we discuss a mechanism to circumvent the `REAL_GET_TASKS` permission, which is required on Android Lollipop to retrieve the list of running applications.

## 3.1 Usage Statistics in a Controlled Scenario

For a first experiment, we set up a local server hosting a website and we launched `tcpdump` to dump all incoming and outgoing TCP packets on this server. Furthermore, we launched the browser application on one of our test devices (a Samsung Galaxy SIII) and retrieved its UID. We navigated to the website hosted on our server and monitored `tcp_snd` and `tcp_rcv` for a period of ten seconds with a sampling frequency of 50 Hz.

Figure 1 illustrates the accumulated TCP packet lengths (left) and the data-usage statistics on the Android smartphone (right). We indicate the outgoing traffic on our server as well as the incoming traffic on the smartphone above the x-axis. Similarly, we indicate the incoming traffic on our server as well as the outgoing traffic on the smartphone below the x-axis. For the sake of readability, we removed measurement samples where the traffic statistics did not change, i.e., we removed consecutive samples where the `tcp_rcv` and `tcp_snd` values did not change. Furthermore, we labeled each TCP packet with the corresponding packet length in both plots. The left plot shows the generated TCP packets according to our website. The first three outgoing packets (1448, 1448, 1417) correspond to the HTML page itself and the following packets (1448, 1448, 1448, 71) correspond to the retrieval of the embedded image. Interestingly, the data-usage statistics on the Android smartphone (right) corresponds to these TCP packet lengths, except for the last two packets (1448, 71), which are observed as one "large" packet (1519=1448+71) instead of two separate packets.

The same observation also holds for the incoming traffic on the server and the outgoing traffic on the smartphone, which are indicated below the x-axis. The corresponding TCP packet lengths can be observed in the outgoing data-usage statistics (366, 364).

The plots in Figure 1 illustrate the observed TCP packet lengths when loading the website for the first time, i.e., without any data being cached. When visiting the website for the second time, the traffic signature slightly changes. More specifically, the second part of the packet sequence (1448, 1448, 1448, 71) is missing as the embedded image is not requested anymore. However, some packet lengths remain the same, regardless of whether the website is cached or not.

**Sampling Frequency.** Zhou et al. [44] reported to be able to observe single TCP packet lengths with a sampling frequency of 10 Hz most of the time. We performed experiments with higher sampling frequencies but also observed the aggregation of multiple TCP packet lengths as one "larger" packet from time to time. A more detailed investigation of specific browser implementations—which is beyond the scope of this paper—might reveal further insights regarding the missed TCP packet lengths and might allow to pick up every single TCP packet properly. Nevertheless, even with some TCP packet lengths being accumulated into one observation, we can successfully exploit this side channel with a sampling frequency between 20 Hz and 50 Hz.

## 3.2 Usage Statistics for Real Websites

In order to investigate the information leakage for real websites, we developed an application that performs the following actions. First, we launch the browser and retrieve its UID. Then, we load three different websites (`google.com`, `facebook.com`, and `youtube.com`) and monitor `tcp_snd` and `tcp_rcv` for a period of ten seconds. The resulting plots can be seen in Figure 2. According to the notion of Jana and Shmatikov [22], these measurements are *stable*, meaning that these observations are similar across visits to the same page, and also *diverse*, meaning that these observations are dissimilar for visits to different pages. Hence, this plot confirms our previous observation that the data-usage statistics can be used to distinguish websites. A similar plot can be obtained from the `tcp_snd` file, i.e., the outbound network traffic, but has been omitted due to reasons of brevity.

## 3.3 Usage Statistics in the Tor Setting

**Background on Tor.** Before we investigate the information leakage of the data-usage statistics in case the network traffic is routed through the Tor network, we start with some background information on Tor. The major design goal of Tor [10] is "to frustrate attackers from linking communication partners" by considering an attacker who can, for instance, observe the network traffic. Therefore, a user runs a so-called onion proxy that is responsible for handling connections from user applications (e.g., the browser), fetching directories (e.g., known onion routers), and establishing circuits (paths) through the network. Such circuits consist of multiple onion routers—which are connected by means of a TLS connection—and are updated periodically. However, establishing such circuits is a costly action that takes some time, which is why multiple TCP streams share one circuit. The onion proxy accepts TCP streams from user applications (browsers) and forwards the data in fixed-size cells (512 bytes) through the TLS connection to the Tor network.

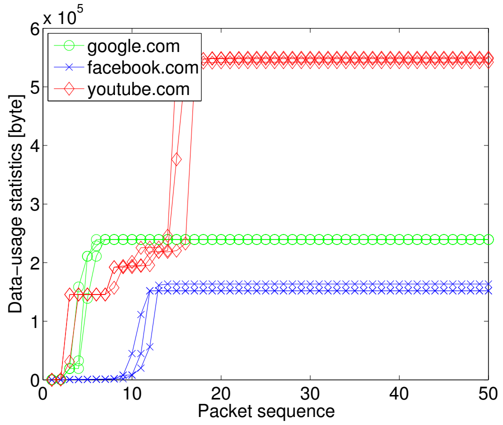

**Figure 2: Data-usage statistics for the inbound traffic of three samples per website.**

**Information Leakage.** In order to investigate the information leakage for traffic routed through the Tor network, we installed the Orweb browser and the corresponding Orbot proxy. The Orweb browser represents the user application and the Orbot proxy represents the onion proxy that handles connections from the Orweb browser and forwards the data to the Tor network. Since websites take longer to load, we increased the time for sampling the data-usage statistics to 20 seconds. As Tor on Android relies on two different applications, i.e., the Orweb browser and the Orbot proxy, we investigated the information leakage for both applications. While the Orweb browser communicates with the Orbot proxy only, the Orbot proxy communicates with the browser as well as the Tor network. Thus, the data-usage statistics for the Orbot proxy are slightly higher. However, both applications revealed the exact same behavior, i.e., the data-usage statistics yield stable and diverse plots, which can be exploited for website fingerprinting attacks. We also installed Firefox 42.0 and configured it to use the Orbot proxy. Repeating the experiment yields the same result, i.e., the data-usage statistics gathered for Firefox allow us to perform website-fingerprinting attacks even if Firefox is configured to route the network traffic through the Tor network. We stress that this is not a vulnerability of Tor or any browser but a fundamental weakness of the Android OS.

Even though data is sent through the Tor network in fixed-size cells (512 bytes), the data-usage statistics leak enough information to perform website fingerprinting attacks. We do not even need to extract complex features as in case of network-based fingerprinting attacks. Instead, the simple yet efficient observation of the data-usage statistics allows us to infer the user's visited websites as if the browser accesses the website directly (cf. Section 3.2).

## 3.4 Usage Statistics for Mobile Connections

The above performed experiments have been carried out with WLAN connections. For the sake of completeness, we also performed experiments with mobile data connections to be sure that we observe the same information leakage when the device is connected, e.g., via the 3G wireless network. The results confirmed our initial observations regarding the data-usage statistics also for mobile connections.

**Table 1: Test devices and configurations**

| Device | OS | Browser/Orbot |
|---|---|---|
| Acer Iconia A510 | Android 4.1.2 | Chrome 44.0 |
| Alcatel One Touch Pop 2 | Android 4.4.4 | Browser 4.4.4 (default browser) |
| Nexus 9 | Android 5.1.1 | Chrome 40.0 |
| Samsung Galaxy SII | Android 2.3.4 | Internet 2.3.4 (default browser) |
| Samsung Galaxy SII | Android 2.3.4 | Orweb 0.7 and Orbot 13.0.4a |
| Samsung Galaxy SII | Android 2.3.4 | Firefox 42.0 and Orbot 13.0.4a |
| Samsung Galaxy SIII | Android 4.3 | Internet 4.3 (default browser) |

### 3.5 API and /proc Support

Table 1 summarizes our test devices and their corresponding configurations. On most of these devices, we accessed the corresponding files within the `/proc` file system to retrieve the data-usage statistics. However, on the Alcatel One Touch Pop 2, the `uid_stat` file does not exist within the `/proc` file system, yet the `TrafficStats` API returned the accumulated bytes received (`getUidRxBytes([uid])`) and transmitted (`getUidTxBytes([uid])`) for a given UID. Similarly, on the Samsung Galaxy SIII, we always retrieved 0 when querying the `TrafficStats` API, but still we were able to read the data-usage statistics from the `/proc` file system.

To summarize our investigations, on some devices an attacker needs to rely on the `/proc` file system, while on others the attacker needs to rely on the `TrafficStats` API. However, all test devices showed the same information leakage through the data-usage statistics.

**REAL_GET_TASKS Permission in Lollipop.** Since Android Lollipop (5.0), the `REAL_GET_TASKS` permission is required to retrieve all running applications via `ActivityManager.getRunningAppProcesses()`. However, one can bypass this permission by retrieving a list of installed applications via `PackageManager.getInstalledApplications()`. The returned information also contains the UID for each of the installed applications. Now, instead of waiting for the browser application to show up in the list returned via `getRunningAppProcesses()`, the malicious application can also wait for the `tcp_rcv` file to be created, which indicates that the application with the given UID has been started. Another alternative to retrieve all running applications is the unprivileged `ps` command. Thus, even on Android Lollipop, our malicious service can be implemented without any suspicious permission and is still able to wait in the background for the browser application to start.

## 4. ADVERSARY MODEL AND SCENARIO

Traditional website fingerprinting attacks consider a *network attacker* who is located somewhere on the victim's network. As illustrated in Figure 3, the adversary observes the encrypted communication between a client and a proxy (or similarly the encrypted communication between the client and the Tor network). In contrast, we consider an attacker who managed to spread a malicious application through a popular app market like *Google Play* or the *Amazon Appstore*. The malicious application running in unprivileged mode represents a *passive* attacker that observes the incoming and outgoing traffic statistics for any target application, e.g., the browser. Figure 4 illustrates this attack scenario for traffic routed through the Tor network. In this case, the user browses the web with the Browser application (e.g., Orweb) and the traffic is routed through the Tor network by means of a dedicated Proxy (e.g., Orbot). However, our attack works analogously in case the browser connects to websites directly. Furthermore, our attack also works against the Proxy application, as has been discussed in Section 3.3.

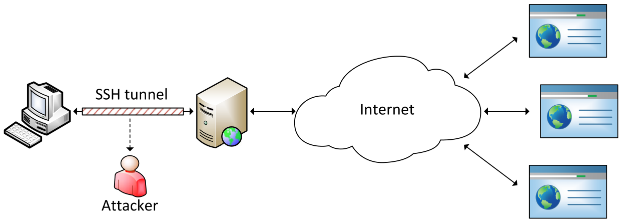

**Figure 3: Traditional website fingerprinting attack considering a network attacker.**

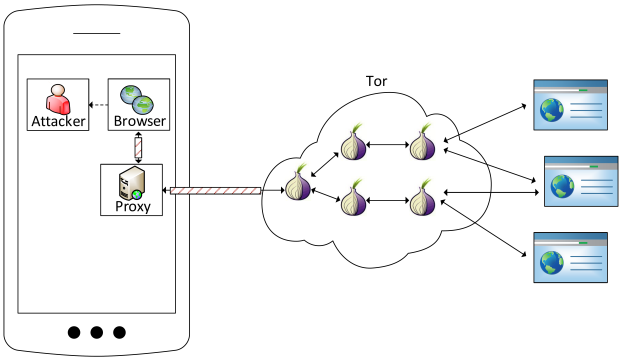

**Figure 4: Client-side website fingerprinting attack exploiting side-channel information.**

According to the notion of Diaz et al. [9], our attacker is passive as it cannot add, drop or change packets. However, this also means that our attacker is lightweight in terms of resource usage as it runs in the background and waits for the browser application to start. Below we describe two possible attack scenarios, one where the training data is gathered on dedicated devices and another one where the attack application gathers the training data directly on the device under attack. Note that the `INTERNET` permission is not required at all due the following reasons. Since Android Marshmallow (6.0), the `INTERNET` permission is granted automatically[3] and below Android 6.0, `ACTION_VIEW`[4] Intents can be used to access the Internet via the browser without this permission.

Since the application neither requires any suspicious permission nor exploits specific vulnerabilities except accesses to publicly readable files and the Android API, the sanity checks performed by app markets (e.g., Google Bouncer) will not detect any malicious behavior. Thus, the application can be spread easily via available app markets. Based on the presented adversary model and the low effort to spread such a malicious application, there is a significantly higher attack potential than in previous fingerprinting attacks. Furthermore, as discussed in Section 1.1, an attacker could combine website fingerprinting attacks with sensor-based keyloggers to launch large-scale identity theft attacks.

[3] `http://developer.android.com/guide/topics/security/normal-permissions.html`

[4] `http://www.leviathansecurity.com/blog/zero-permission-android-applications`

## 4.1 Possible Attack Scenarios

In order to exploit the information leakage, we consider two different attack scenarios.

**Scenario 1.** For this scenario we assume that the malicious application does not capture the required training data on the device itself. Instead, a more powerful adversary gathers the training data on specifically deployed devices. The application only waits for the browser to start, gathers the traffic information, and sends the gathered data to the remote server that infers the visited websites. In order to match the device name of the attacked device with the training devices, the `android.os.Build` API can be used.

**Scenario 2.** For this scenario we assume that the malicious application captures the required training data directly on the device. Therefore, it triggers the browser to load a list of websites, one after each other, via the `ACTION_VIEW` Intent. While the browser loads the website, the malicious application captures the traffic statistics from `tcp_rcv` and `tcp_snd`, which then acts as training data. After collecting the required training data, the application waits in the background until the unsuspecting user opens the browser and starts browsing the web. Then, the application gathers the traffic statistics from `tcp_rcv` and `tcp_snd` again, and matches the collected information against the previously established training data to infer the visited websites.

A technicality that needs to be solved in case of scenario 2 is that users should not notice the gathering of the training data. For that purpose, we note that Zhou et al. [44] demonstrated the possibility to (1) wait for the screen to dim before launching the browser, and (2) to close the browser after loading the website. Thereby, the user does not observe any suspicious behavior, even though the application launches the browser application in the foreground. However, the main drawback of this approach is that the device might switch to sleep mode and pause all activities, which means that gathering the training data takes some time.

## 4.2 Assumptions

According to Wang et al. [38], existing fingerprinting attacks rely on two assumptions. We briefly summarize these assumptions and argue why our attack approach is more realistic than existing fingerprinting attacks.

1. *The attacker knows the start and the end of a packet trace.* This assumption is based on the observation that users take some time to load the next webpage. We justify this assumption by arguing that we are able to determine when the browser starts. Thus, we are able to observe the first trace. Afterwards, we assume that the user takes some time to load the next page.
2. *The client does not perform any other activity that can be confused with page-loading activities, for example, a file download.* Hayes and Danezis [18] pointed out that it is highly unlikely that an attacker will be able to gather traffic information without background traffic, which limits the applicability of existing website fingerprinting attacks. However, Android captures the data-usage statistics on a per-application basis and, thus, our approach is invariant to network activities of other applications. For instance, our attack also works in case an e-Mail app, WhatsApp, or any other app fetches updates in the background while the user browses the web. In contrast, it is highly unlikely that the network traffic on the wire does not contain any background traffic.

Another thing that needs to be clarified is the browser's caching behavior. For our experiments, we do not clean the cache before loading a page as we assume that adversaries might be interested in identifying frequently visited websites of a user. If users frequently visit specific websites, then these sites are most probably already cached. Still, specific parts of the TCP packets are equal between cached and non-cached pages, as has been discussed in Section 3.1. Our experiments with the Orweb browser use the default settings, meaning that caching of websites is disabled. Thus, we provide insights for both settings of the caching behavior.

# 5. ATTACK DESCRIPTION

Based on the presented adversary model and attack scenarios, we now describe the attack in more detail. First, we discuss how to gather the required traffic statistics for a set of monitored websites. Afterwards, we describe the employed classifier to infer the visited websites.

## 5.1 Gathering Traffic Signatures

The list of monitored websites might, for example, be chosen according to specific interests like political interests, sexual orientation, illnesses, or websites that are supposed to be blocked. For our evaluation, we decided to use popular websites among different categories according to `alexa.com`. The fact that Tor browsers, e.g., Orweb, do not cache pages, leads to the realistic scenario that an adversary wants to monitor landing pages (cf. [18]). Thus, we consider our chosen setting for the evaluation as being realistic.

Algorithm 1 summarizes the basic steps to establish the required training data denoted as traffic signature database $T$. The algorithm is given a list of monitored websites $W$, the desired number of samples per website $n$, a profiling time $\tau$, and a sampling frequency $f$. For each website $w_i \in W$, the algorithm loads this website within the browser. While the browser application loads the website $w_i$, we gather the data-usage statistics $f$ times per second for a period of $\tau$ seconds. Each tuple $(w_i, t_i)$, which is denoted as one *sample* for a specific website $w_i$, is added to $T$. These steps are repeated until $n$ samples have been gathered for each website. Finally, the algorithm returns the traffic signature database $T$.

**Algorithm 1:** Gathering training samples.

```
Input: List of monitored websites $W$, number of
       samples per website $n$, profiling time $\tau$,
       sampling frequency $f$
Output: Traffic signature database $T$

Launch browser application and retrieve its UID
repeat $n$ times
    foreach website $w_i$ in $W$ do
        simultaneously
        Launch website $w_i$ in browser
        while profiling time $\tau$ not passed do
            $f$ times per second
            read tcp_rcv and append to $t_{IN}$
            read tcp_snd and append to $t_{OUT}$
        end
        $t_i \leftarrow \{t_{IN}, t_{OUT}\}$
        Add tuple $(w_i, t_i)$ to $T$
    end
end
```

## 5.2 Classification

The traffic signature database $T$ requires only minor preprocessing before the actual classification. More specifically, we removed samples of websites that did not load, i.e., we removed tuples $(w_i, t_i)$ from $T$ where all entries in $t_i$ are 0. Furthermore, if $n-1$ samples for a specific website are removed, we remove the remaining sample as well. We justify this as follows. If this single remaining sample of a specific website is used for training, then it cannot be used for evaluation purposes. Similarly, if we do not have a single sample for training, then this site will never be classified correctly.

We use the Jaccard index as a metric to determine the similarity between two websites. In case of our measurement samples, the Jaccard index as defined in Equation 1 compares two traces $t_1$ and $t_2$ based on unique and distinguishable packet lengths.

$$\text{Jaccard}(t_1, t_2) = \frac{|t_1 \cap t_2|}{|t_1 \cup t_2|} \tag{1}$$

We consider the traces of the inbound and outbound traffic separately. Hence, our classifier aims to find the maximum similarity for a given trace $t_A = \{t_{IN_A}, t_{OUT_A}\}$ compared to all traces $t_i = \{t_{IN_i}, t_{OUT_i}\}$ within the previously established traffic signature database $T$. We illustrate this similarity measure in Equation 2.

$$\begin{aligned} &\text{Sim}(t_A = \{t_{IN_A}, t_{OUT_A}\}, t_i = \{t_{IN_i}, t_{OUT_i}\}) = \\ &\quad \text{Jaccard}(t_{IN_A}, t_{IN_i}) + \text{Jaccard}(t_{OUT_A}, t_{OUT_i}) \end{aligned} \tag{2}$$

Based on this similarity metric, we implemented our classifier as outlined in Algorithm 2. The algorithm is given a list of monitored websites $W$, a traffic signature database $T$, and the signature $t$ to be classified. As $T$ contains multiple samples $(w_i, t_i)$ for one website, we compute the similarity of $t$ with all these traffic signatures $t_j$ with $1 \leq j \leq n$ corresponding to a specific website $w_i$. Afterwards, we compute the average similarity with all these traces for this specific website $w_i$. Finally, we return the website $w_i$ with the highest average similarity $s_i$ compared to the given trace $t$.

**Algorithm 2:** Classification algorithm.

```
Input: List of monitored websites $W$, traffic signature
       database $T$, traffic signature $t$
Output: Website $w$

foreach website $w_i$ in $W$ do
    Retrieve all samples $(w_i, t_1), \dots, (w_i, t_n) \in T$
    $s_i = \text{avg}(\text{Sim}(t_1, t), \dots, \text{Sim}(t_n, t))$
    Add tuple $(w_i, s_i)$ to $S$
end
Return $(w_i, s_i) \in S$, s.t. $s_i$ is maximized
```

Compared to network-based fingerprinting attacks, our attack relies on a simple yet efficient classifier. We do not need a dedicated training phase that requires several hundred CPU hours for some network-based fingerprinting attacks (cf. [38, 39]). Still, the testing time of our classifier is comparable to existing fingerprinting attacks and yields highly accurate results as will be discussed in Section 6.5.

# 6. EVALUATION AND RESULTS

We now evaluate the classification rate for a standard browser application and continue with a setting where the traffic is routed through the Tor network. We also investigate how the classification rate decreases over time and we evaluate the scalability of our attack for larger sets of monitored websites. Finally, we compare our results to related work. All experiments in this section have been performed with data-usage statistics captured via WLAN connections.

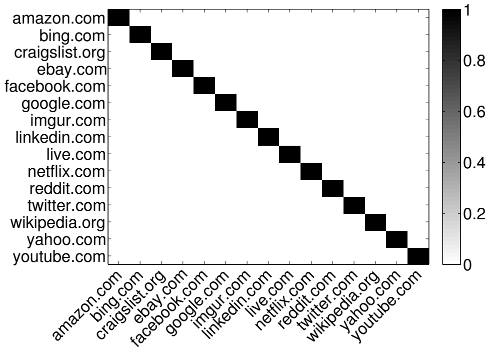

**Figure 5: Confusion matrix for the 15 most popular websites in the US.**

**Table 2: Google websites that have been merged**

| | | |
|---|---|---|
| google.co.in | google.co.jp | google.de |
| google.co.uk | google.com.br | google.fr |
| google.ru | google.it | google.es |
| google.ca | google.com.mx | google.com.hk |
| google.com.tr | google.co.id | google.pl |
| google.com.au | google.co.kr | googleadservices.com |

## 6.1 Intra-Day Classification Rate

For our first experiment, we took the 15 most popular websites in the US according to `alexa.com` and we gathered $n = 5$ samples for each of these websites to establish the signature database $T$. In order to estimate the performance of our classifier, we performed a leave-one-out cross validation. Thus, for each sample $(w_i, t_i) \in T$, we removed this sample (one at a time) from the traffic signature database $T$, and called the classification algorithm (Algorithm 2) with the traffic signature database $T \setminus (w_i, t_i)$ and the traffic signature $t_i$ to be classified.

Figure 5 illustrates the resulting confusion matrix. We indicate the ground truth along the y-axis and the inferred website along the x-axis. Since each of the five page visits to each of these 15 websites has been classified correctly, we achieve a classification rate of 100%. More specifically, each sample $(w_i, t_i)$ has been classified correctly considering the traffic signature database $T \setminus (w_i, t_i)$ for training the classifier.

If we have a look at the 100 most popular websites globally, then we observe a classification rate of 89% for a total of 500 page visits. After further investigations of the resulting confusion matrix, we noticed that several misclassifications occur because `google*.*` pages have been misclassified among each other. For example, `google.es` has been misclassified as either `google.fr`, `google.it`, or `google.pl`. Nevertheless, we do not aim for a detailed classification of different Google domains and, hence, we merged all websites as shown in Table 2 to be classified as `google.com`.

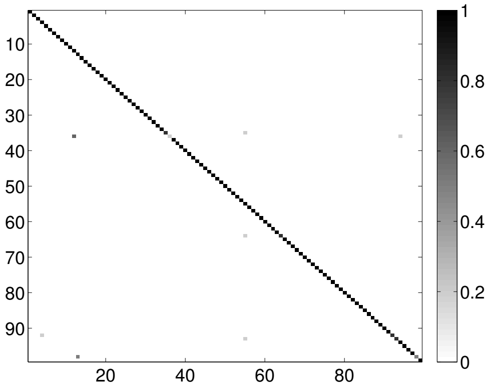

**Figure 6: Confusion matrix for the 100 most popular websites globally with *google*.** pages merged.**

Performing the classification again, we achieve a classification rate of 98% for these 500 page visits. The corresponding confusion matrix can be seen in Figure 6. Merging these Google websites leads to a total of 9 misclassified websites among these 500 page visits, with `mail.ru` at index 36 being the most commonly misclassified website (4 times).

## 6.2 Classification Rate for Tor

We also evaluated our attack for traffic routed through Tor. Therefore, we gathered $n = 5$ samples for the top 100 websites in the US by capturing the data-usage statistics of the Orweb browser, but we used a profiling time of 20 seconds as websites accessed via Tor take more time to load. Again, we performed a leave-one-out cross validation resulting in a classification rate of 95% for these 500 page visits.

If we instruct the classifier to return a set of $k$ possible websites, which are sorted according to their probability for being the correct one, then the success rate steadily increases with the number of websites $k$ taken into consideration. As can be seen in Figure 7, taking the two most probable websites ($k = 2$) into consideration, we achieve a classification rate of 97% and taking the three most probable websites ($k = 3$) into consideration yields a classification rate of 98%. Similar classification rates can be observed for the standard Android browser where the traffic is not routed through Tor. Although the standard browser yields slightly better classification rates, we did not observe significant differences between the classification rates when accessing websites directly and when accessing websites via Tor.

**Security Implications.** Even though browsers (e.g., Orweb or "private/incognito" modes) do not store the browsing history and the network traffic is protected against specific attacks while being routed through the Tor network, an unprivileged application can infer the user's browsing behavior for monitored websites. Thus, the `READ_HISTORY_BOOKMARKS` permission does not protect the user's privacy and even routing the network traffic through Tor provides a false sense of privacy for Android users. Furthermore, given the fact that Orweb disables JavaScript and Flash by default, users do not use Orweb to access sites that heavily rely on these techniques. Hence, attackers explicitly targeting Tor users can significantly reduce the set of monitored websites, which increases the success rate. However, we do not blame the browsers for these security implications but the Android OS.

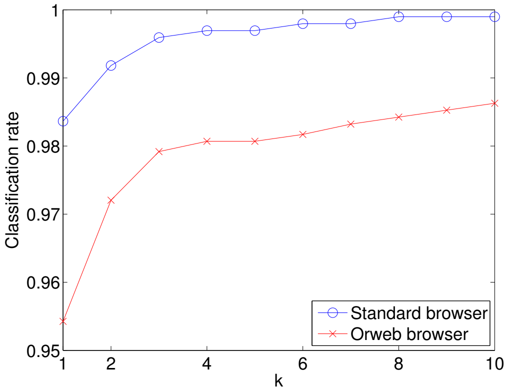

**Figure 7: Classification rates considering the $k$ most probable websites returned from the classifier.**

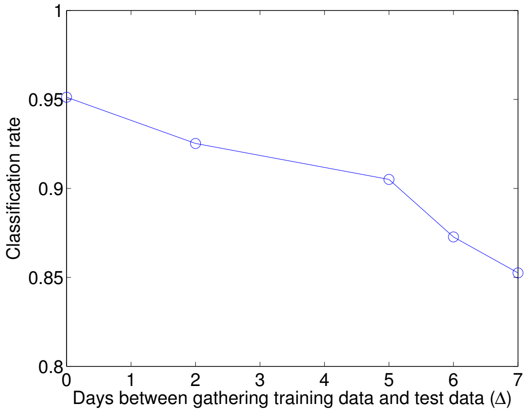

**Figure 8: Decreasing accuracy for samples captured $\Delta$ days after the training data.**

## 6.3 Inter-Day Classification Rate

We also performed experiments with an outdated training set. In order to do so, we used Orweb to collect measurement samples for the top 100 websites in the US a few days after gathering the training data. The evaluation in Figure 8 shows that the accuracy decreases rather slowly. For test data that has been gathered on the same day as the training data ($\Delta = 0$), 95% of all page visits can be inferred. Using the same data to classify 500 page visits captured two days later ($\Delta = 2$) still yields a classification accuracy of 93%. Testing measurement samples captured after five days ($\Delta = 5$) yields a classification accuracy of 91%. Even data gathered one week later ($\Delta = 7$) achieves a classification rate of 85%. Hence, even slightly outdated training data allows to infer websites accurately.

This slowly decreasing classification rate allows an adversary to keep the traffic signature database $T$ for some time, meaning that the adversary does not need to update the database for the monitored websites on a daily basis. Thereby, the attacker's effort and workload can be reduced significantly, which leads to more practical attacks. Furthermore, this also indicates that the training samples can be gathered after the actual attack samples, which represents another advantage for the attacker (cf. [27]).

**Table 3: Websites with the highest number of misclassifications in the inter-day setting**

| $\Delta$ | Website | # misclassifications |
|---|---|---|
| 2 days | `ask.com` | 5 times |
| 2 days | `twitch.tv` | 5 times |
| 2 days | `cnn.com` | 3 times |
| 5 days | `bbc.com` | 5 times |
| 5 days | `indeed.com` | 5 times |
| 5 days | `nytimes.com` | 5 times |
| 5 days | `twitch.tv` | 5 times |
| 5 days | `espn.go.com` | 4 times |
| 6 days | `bbc.com` | 5 times |
| 6 days | `indeed.com` | 5 times |
| 6 days | `nytimes.com` | 5 times |
| 6 days | `twitch.tv` | 5 times |
| 6 days | `espn.go.com` | 4 times |
| 7 days | `bbc.com` | 5 times |
| 7 days | `bleacherreport.com` | 5 times |
| 7 days | `indeed.com` | 5 times |
| 7 days | `nytimes.com` | 5 times |
| 7 days | `twitch.tv` | 5 times |
| 7 days | `xfinity.com` | 5 times |
| 7 days | `homedepot.com` | 4 times |

Table 3 shows the worst websites (according to their classification accuracy) in case the attack samples are captured $\Delta$ days after the training data. As news sites tend to change frequently, it is not surprising that the classification rate decreases for such websites. Based on this observation, an attacker can selectively rebuild the training set for frequently changing websites like news portals, while the training data for more static websites can be kept for a longer period.

We point out that we did not employ any measures to ensure that we exit Tor with a specific country IP. Instead, every time we gathered new test samples, we reconnected to the Tor network by restarting the Orbot proxy. Nevertheless, the classification rates indicate a high success rate even with a training set that is not completely up to date.

## 6.4 Scalability for Larger World Sizes

In order to investigate the scalability of our attack for a larger set of monitored websites, we consider the top 500 websites. However, we did not route the network traffic through Tor as this would have taken significantly longer. Our results indicate that we are able to classify 97% of 2 500 page visits out of a set of 500 monitored websites.

## 6.5 Comparison with Related Work

As our attack only requires an unprivileged Android application, it is easier to perform than wiretapping attacks where the attacker observes TCP packets on the victim's network. Considering the ease of applicability, the computational performance of the classifier, and the classification rates, our attack outperforms existing fingerprinting attacks.

Table 4 provides a comprehensive comparison of website fingerprinting attacks in the closed-world setting. For each attack, we present the attacker or the exploited information in column 2. Column 3 indicates the caching behavior of the browser. Column 4 shows the employed classifier. For the sake of brevity, we do not list the exact features that are used for classification purposes, but we refer the interested reader to the corresponding works. Column 5 shows the attacked countermeasure, where "none" refers to no specific countermeasure, "SSH tunnel" means that the client hides its network traffic through a proxy where the encrypted communication between the client and the proxy is observed, and "Tor" means that the traffic is routed through the anonymity network Tor. As most website fingerprinting attacks consider a closed-world setting, we state the number of monitored websites in column 6. Furthermore, we indicate the accuracy within the last column.

The only works in Table 4 that exploit client-side side-channel information are the work of Jana and Shmatikov [22] as well as ours. To be more precise, Jana and Shmatikov do not exploit information that relates to the TCP packet lengths. Instead, they exploit the memory footprint of the browser while rendering the monitored website. However, our approach of exploiting the data-usage statistics allows a significantly more accurate inference of visited websites.

Compared to wiretapping attacks against the anonymity network Tor, we observe that our approach of exploiting client-side information leaks yields mostly better results. Though, the works of Wang and Goldberg [39] and Wang et al. [38] achieve a rather similar classification accuracy. For instance, on a set of 100 monitored websites, they achieve a classification accuracy of 91% [39] and 95% [38], respectively. However, training their classifiers, which are based on the optimal string alignment distance and a weighted k-NN, takes a significant amount of time (608 000 CPU seconds [39]). This, however, is not feasible for every attacker. As our classifier does not require a training phase, we significantly reduce the computational effort. Our test algorithm scales linearly with the number of monitored websites and the corresponding samples, i.e., $\mathcal{O}(|W| \cdot n)$. A naive implementation of our classifier (Algorithm 2)—without any specific optimizations—in Matlab takes a testing time of about 0.4 seconds on an Intel Core i7-5600U CPU for a set of 100 monitored websites and 5 samples per website. This testing time is comparable to the testing time of 0.7 seconds reported by Wang and Goldberg [39], but in contrast to their work we do not require an expensive training phase.

**Wiretapping vs. Side-Channel Observations.** The subtle difference between observing traffic information on the wire and the side-channel information of the data-usage statistics requires further considerations. In the wiretapping scenario it is impossible to miss single packets, whereas in in the side-channel scenario the attacker might miss single packets which are then observed as one "large" packet due to the accumulation of TCP packets within the data-usage statistics. However, wiretapping attacks against Tor need to rely on the observation of padded packets only, whereas we also observe unpadded packets due to the separation of the browser and proxy application on the smartphone.

To summarize this comparison, we consider it easier to deploy a malicious application than to wiretap the victim's network, but we trade the exact observation of single TCP packet lengths (or padded packets in case of Tor) for a slightly less accurate side-channel observation. The most significant advantage of our attack is that it only requires an unprivileged Android application and a simple (yet efficient) classifier. Thus, in contrast to wiretapping attacks, the presented attack can be deployed on a large scale. In addition, our client-side attack overcomes many limitations and assumptions of network-based fingerprinting attacks that are considered unrealistic, i.e., it is invariant to background traffic and does not require expensive training phases.

Table 4: Comparison of website fingerprinting attacks in the closed-world setting

| Work | Exploited information | Caching | Classifier | Countermeasure | # websites | Classification rate |
|---|---|---|---|---|---|---|
| Ours | Client-side data-usage statistics | Enabled | Jaccard index | None | 500 | **97%** |
| Jana and Shmatikov [22] | Client-side memory footprint | Enabled?[a] | Jaccard index | None | 100 | 35% |
| Cai et al. [6] | TCP packets captured via tshark | Disabled | Damerau-Levenshtein distance | SSH tunnel | 100 | 92% |
| Herrmann et al. [20] | Client-side tcpdump | Disabled | Naive-Bayes classifier | SSH tunnel | 775 | 96% |
| Liberatore and Levine [27] | Client-side tcpdump | Disabled | Jaccard index | SSH tunnel | 500 | 79% |
| Liberatore and Levine [27] | Client-side tcpdump | Disabled | Naive-Bayes classifier | SSH tunnel | 500 | 75% |
| Ours | Client-side data-usage statistics | Disabled | Jaccard index | Tor | 100 | **95%** |
| Herrmann et al. [20] | Client-side tcpdump | Disabled | Multinomial Naive-Bayes classifier | Tor | 775 | 3% |
| Cai et al. [6] | TCP packets captured via tshark | Disabled | Damerau-Levenshtein distance | Tor | 100 | 84% |
| Panchenko et al. [35] | Client-side tcpdump | Disabled | Support vector machines | Tor | 775 | 55% |
| Wang and Goldberg [39] | TCP packets[b] | Disabled | Optimal string alignment distance[c] | Tor | 100 | 91% |
| Wang and Goldberg [39] | TCP packets[b] | Disabled | Levenshtein distance | Tor | 100 | 70% |
| Wang et al. [38] | TCP packets[d] | Disabled | k-nearest neighbour[e] | Tor | 100 | 95% |

[a] We are not sure whether caching has been disabled for the Android browser.
[b] They parsed TCP packets to obtain the underlying Tor cells but did not specify how to obtain the TCP packets.
[c] Training took 608 000 CPU seconds.
[d] They used the same data set as in [39].
[e] The computational complexity of the training phase is similar to [39] (cf. footnote c).

# 7. DISCUSSION OF COUNTERMEASURES

We now provide an overview of existing countermeasures. However, most of these countermeasures have been proposed to mitigate wiretapping attacks and, thus, we discuss the relevance of these defense mechanisms against our attack.

## 7.1 Existing Countermeasures

**Traffic Morphing.** Wright et al. [40] suggested traffic morphing, which requires the cooperation of the target web server or proxy as well as the browser. Each packet from a website is padded or split in such a way that the traffic information of the actual website matches the traffic information of a different website. As a result, an attacker observing the traffic information will most likely misclassify this website.

**HTTPOS.** A browser-based defense mechanism denoted as HTTP or HTTPS with Obfuscation (HTTPOS) has been proposed by Luo et al. [29]. HTTPOS focuses on changing packet sizes and packet timings, which can be done, for instance, by adding bytes to the referer header or by using the HTTP range option to fetch specific portions of websites.

**BuFLO.** Dyer et al. [11] presented Buffered Fixed-Length Obfuscator (BuFLO) that sends fixed-length packets at fixed intervals for a fixed amount of time. While the authors claim that BuFLO significantly reduces the attack surface, it is rather inefficient in terms of bandwidth overhead. Again, BuFLO requires the cooperation of the involved proxies. Cai et al. [4] proposed an extension denoted as Congestion-Sensitive Buffered Fixed-Length Obfuscator (CS-BuFLO).

**Glove.** Nithyanand et al. [32] proposed Glove, which tries to cluster similar websites according to pre-selected features. Based on these clusters, transcripts of packet sizes (denoted as super-traces) are computed, which are later transmitted whenever a page in the corresponding cluster is loaded. This super-trace is obtained by inserting, merging, splitting, and delaying packets. The major drawback of Glove is that the traces must be updated regularly, which is rather expensive.

## 7.2 Discussion

The above mentioned countermeasures aim at preventing website fingerprinting attacks. Nevertheless, network-level defenses do not provide effective countermeasures against client-side attacks. For instance, if the traffic is routed through the Tor network, then the traffic might be protected on the network. However, unless the browser itself is actively involved in these defense mechanisms, these countermeasures do not provide protection against client-side attackers. We demonstrated this by exploiting the data-usage statistics of browser applications that route the traffic through the Tor network. Furthermore, application-level defenses like HTTPOS might prevent such attacks at first glance, but Cai et al. [4] already demonstrated that a network attacker can circumvent this countermeasure. Besides, many network-level defenses add a significant overhead in terms of bandwidth and data consumption, which is impractical for mobile devices with a limited data plan. We consider further investigations regarding the effectiveness of countermeasures against client-side attacks as an interesting open research problem. Furthermore, new proposals for countermeasures should consider mobile devices.

**Client-Side Countermeasures.** Besides these proposed countermeasures, which mostly target network-level attackers, fixing fundamental design flaws of Android should be considered as absolutely necessary. Zhou et al. [44] suggested two permission-based approaches. The first one is a new permission that allows applications to monitor the data-usage statistics. The second one is to let applications define how data-usage statistics should be published. We, however, do not consider these approaches as viable countermeasures for the following reasons. First, many users either do not pay attention to the requested permissions or they do not understand the meaning of these permissions (cf. [13, 25]). Second, the permission system also confuses developers which leads to overprivileged applications [12]. Besides, developers might not be aware that the data-usage statistics of their application leaks sensitive information and, thus, should impose restrictions on how to publish these statistics.

A more general approach to prevent such side-channel attacks has been suggested by Zhang et al. [43]. The basic idea of their approach is that an application (*App Guardian*) pauses/stops suspicious background processes when the application to be protected (principal) is executed. This idea sounds quite appealing but still struggles with unsolved issues like a proper identification of malicious processes.

A first solution to defend against such attacks would be to update these statistics according to a more coarse-grained granularity. We stress that data-usage statistics capturing single TCP packet lengths represents a significant threat. Updating data-usage statistics in a more coarse-grained interval, e.g., on a daily basis, should suffice for users to keep an eye on their data consumption. Future work might come up with more advanced countermeasures and the above outlined approach of *App Guardian* [43] definitely follows the right direction towards the prevention of such attacks. Still, we urge OS developers to address this issue immediately.

# 8. CONCLUSION

In this work, we investigated a new type of client-side website fingerprinting attack that exploits the data-usage statistics published by Android. We argue that incognito/private browsing modes, the `READ_HISTORY_BOOKMARKS` permission, and even routing the network traffic through Tor provides a false sense of privacy for smartphone users. Even though the browser itself does not store any information about visited websites and the traffic is protected while being routed through Tor, the data-usage statistics leak sensitive information. We demonstrated that any application can accurately infer a user's visited websites without any suspicious permission. Hence, the `READ_HISTORY_BOOKMARKS` permission—which is supposed to protect a user's browsing behavior—is actually irrelevant as it does not provide protection.

Compared to existing website fingerprinting attacks, our attack can be deployed significantly easier and allows for more accurate classifications of websites. The ease of applicability allows even less sophisticated attackers to perform accurate website fingerprinting attacks on a large scale, which clearly proves the immense threat arising from this information leak. As a user's browsing behavior reveals rather sensitive information, we urge the need to address this issue on the operating system level. Furthermore, due to the simple (yet accurate) classification algorithm, real-time detection of the user's browsing behavior in combination with sensor-based keyloggers allows for large-scale identity theft attacks which must be prevented by all means.

## Acknowledgment

This work has been supported by the Austrian Research Promotion Agency (FFG) and the Styrian Business Promotion Agency (SFG) under grant number 836628 (SeCoS) and in part by the European Commission through the FP7 program under project number 610436 (project MATTHEW).

# 9. REFERENCES

[1] A. J. Aviv, B. Sapp, M. Blaze, and J. M. Smith. Practicality of Accelerometer Side Channels on Smartphones. In *Annual Computer Security Applications Conference – ACSAC 2012*, pages 41–50. ACM, 2012.

[2] G. D. Bissias, M. Liberatore, D. Jensen, and B. N. Levine. Privacy Vulnerabilities in Encrypted HTTP Streams. In *Privacy Enhancing Technologies – PET 2005*, volume 3856 of *LNCS*, pages 1–11. Springer, 2005.

[3] L. Cai and H. Chen. TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion. In *USENIX Workshop on Hot Topics in Security – HotSec*. USENIX Association, 2011.

[4] X. Cai, R. Nithyanand, and R. Johnson. CS-BuFLO: A Congestion Sensitive Website Fingerprinting Defense. In *Workshop on Privacy in the Electronic Society – WPES 2014*, pages 121–130. ACM, 2014.

[5] X. Cai, R. Nithyanand, T. Wang, R. Johnson, and I. Goldberg. A Systematic Approach to Developing and Evaluating Website Fingerprinting Defenses. In *Conference on Computer and Communications Security – CCS 2014*, pages 227–238. ACM, 2014.

[6] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson. Touching from a Distance: Website Fingerprinting Attacks and Defenses. In *Conference on Computer and Communications Security – CCS 2012*, pages 605–616. ACM, 2012.

[7] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *IEEE Symposium on Security and Privacy – S&P 2010*, pages 191–206. IEEE Computer Society, 2010.

[8] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde. Analyzing Android Encrypted Network Traffic to Identify User Actions. *IEEE Transactions on Information Forensics and Security*, 11:114–125, 2016.

[9] C. Díaz, S. Seys, J. Claessens, and B. Preneel. Towards Measuring Anonymity. In *Privacy Enhancing Technologies – PET 2002*, volume 2482 of *LNCS*, pages 54–68. Springer, 2002.

[10] R. Dingledine, N. Mathewson, and P. F. Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium 2004*, pages 303–320. USENIX, 2004.

[11] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail. In *IEEE Symposium on Security and Privacy – S&P 2012*, pages 332–346. IEEE Computer Society, 2012.

[12] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Conference on Computer and Communications Security – CCS 2011*, pages 627–638. ACM, 2011.

[13] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *Symposium On Usable Privacy and Security – SOUPS 2012*, page 3. ACM, 2012.

[14] E. W. Felten and M. A. Schneider. Timing Attacks on Web Privacy. In *Conference on Computer and Communications Security – CCS 2000*, pages 25–32. ACM, 2000.

[15] X. Gong, N. Borisov, N. Kiyavash, and N. Schear. Website Detection Using Remote Traffic Analysis. In *Privacy Enhancing Technologies – PET 2012*, volume 7384 of *LNCS*, pages 58–78. Springer, 2012.

[16] X. Gong, N. Kiyavash, and N. Borisov. Fingerprinting Websites Using Remote Traffic Analysis. In *Conference on Computer and Communications Security – CCS 2010*, pages 684–686. ACM, 2010.

[17] D. Gruss, D. Bidner, and S. Mangard. Practical Memory Deduplication Attacks in Sandboxed Javascript. In *European Symposium on Research in Computer Security – ESORICS 2015*, volume 9326 of *LNCS*, pages 108–122. Springer, 2015.

[18] J. Hayes and G. Danezis. Better Open-World Website Fingerprinting. *CoRR*, abs/1509.00789, 2015.

[19] G. He, M. Yang, X. Gu, J. Luo, and Y. Ma. A Novel Active Website Fingerprinting Attack Against Tor Anonymous System. In *Computer Supported Cooperative Work in Design – CSCWD 2014*, pages 112–117. IEEE, 2014.

[20] D. Herrmann, R. Wendolsky, and H. Federrath. Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naïve-Bayes Classifier. In *Cloud Computing Security Workshop – CCSW*, pages 31–42. ACM, 2009.

[21] A. Hintz. Fingerprinting Websites Using Traffic Analysis. In *Privacy Enhancing Technologies – PET 2002*, volume 2482 of *LNCS*, pages 171–178. Springer, 2002.

[22] S. Jana and V. Shmatikov. Memento: Learning Secrets from Process Footprints. In *IEEE Symposium on Security and Privacy – S&P 2012*, pages 143–157. IEEE Computer Society, 2012.

[23] A. Janc and L. Olejnik. Web Browser History Detection as a Real-World Privacy Threat. In *European Symposium on Research in Computer Security – ESORICS 2010*, volume 6345 of *LNCS*, pages 215–231. Springer, 2010.

[24] M. Juárez, S. Afroz, G. Acar, C. Díaz, and R. Greenstadt. A Critical Evaluation of Website Fingerprinting Attacks. In *Conference on Computer and Communications Security – CCS 2014*, pages 263–274. ACM, 2014.

[25] P. G. Kelley, S. Consolvo, L. F. Cranor, J. Jung, N. M. Sadeh, and D. Wetherall. A Conundrum of Permissions: Installing Applications on an Android Smartphone. In *Financial Cryptography – FC 2012*, volume 7398 of *LNCS*, pages 68–79. Springer, 2012.

[26] B. Liang, W. You, L. Liu, W. Shi, and M. Heiderich. Scriptless Timing Attacks on Web Browser Privacy. In *Dependable Systems and Networks – DSN 2014*, pages 112–123. IEEE, 2014.

[27] M. Liberatore and B. N. Levine. Inferring the Source of Encrypted HTTP Connections. In *Conference on Computer and Communications Security – CCS 2006*, pages 255–263. ACM, 2006.

[28] L. Lu, E. Chang, and M. C. Chan. Website Fingerprinting and Identification Using Ordered Feature Sequences. In *European Symposium on Research in Computer Security – ESORICS 2010*, volume 6345 of *LNCS*, pages 199–214. Springer, 2010.

[29] X. Luo, P. Zhou, E. W. W. Chan, W. Lee, R. K. C. Chang, and R. Perdisci. HTTPOS: Sealing Information Leaks with Browser-side Obfuscation of Encrypted Flows. In *Network and Distributed System Security Symposium – NDSS 2011*. The Internet Society, 2011.

[30] B. Miller, L. Huang, A. D. Joseph, and J. D. Tygar. I Know Why You Went to the Clinic: Risks and Realization of HTTPS Traffic Analysis. In *Privacy Enhancing Technologies – PET 2014*, volume 8555 of *LNCS*, pages 143–163. Springer, 2014.

[31] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury. Tapprints: Your Finger Taps Have Fingerprints. In *Mobile Systems – MobiSys 2012*, pages 323–336. ACM, 2012.

[32] R. Nithyanand, X. Cai, and R. Johnson. Glove: A Bespoke Website Fingerprinting Defense. In *Workshop on Privacy in the Electronic Society – WPES 2014*, pages 131–134. ACM, 2014.

[33] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *Conference on Computer and Communications Security – CCS 2015*, pages 1406–1418. ACM, 2015.

[34] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang. ACCessory: Password Inference Using Accelerometers on Smartphones. In *Mobile Computing Systems and Applications – HotMobile 2012*, page 9. ACM, 2012.

[35] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website Fingerprinting in Onion Routing Based Anonymization Networks. In *Workshop on Privacy in the Electronic Society – WPES 2011*, pages 103–114. ACM, 2011.

[36] R. Spreitzer. PIN Skimming: Exploiting the Ambient-Light Sensor in Mobile Devices. In *Security and Privacy in Smartphones & Mobile Devices – SPSM@CCS*, pages 51–62. ACM, 2014.

[37] Q. Sun, D. R. Simon, Y. Wang, W. Russell, V. N. Padmanabhan, and L. Qiu. Statistical Identification of Encrypted Web Browsing Traffic. In *IEEE Symposium on Security and Privacy – S&P 2002*, pages 19–30. IEEE Computer Society, 2002.

[38] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg. Effective Attacks and Provable Defenses for Website Fingerprinting. In *USENIX Security Symposium 2014*, pages 143–157. USENIX Association, 2014.

[39] T. Wang and I. Goldberg. Improved Website Fingerprinting on Tor. In *Workshop on Privacy in the Electronic Society – WPES 2013*, pages 201–212. ACM, 2013.

[40] C. V. Wright, S. E. Coull, and F. Monrose. Traffic Morphing: An Efficient Defense Against Statistical Traffic Analysis. In *Network and Distributed System Security Symposium – NDSS 2009*. The Internet Society, 2009.

[41] Z. Xu, K. Bai, and S. Zhu. TapLogger: Inferring User Inputs On Smartphone Touchscreens Using On-board Motion Sensors. In *Security and Privacy in Wireless and Mobile Networks – WISEC 2012*, pages 113–124. ACM, 2012.

[42] K. Zhang and X. Wang. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In *USENIX Security Symposium 2009*, pages 17–32. USENIX Association, 2009.

[43] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. Leave Me Alone: App-Level Protection against Runtime Information Gathering on Android. In *IEEE Symposium on Security and Privacy – S&P 2015*, pages 915–930. IEEE Computer Society, 2015.

[44] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, Location, Disease and More: Inferring Your Secrets from Android Public Resources. In *Conference on Computer and Communications Security – CCS 2013*, pages 1017–1028. ACM, 2013.