# No Free Charge Theorem: A Covert Channel via USB Charging Cable on Mobile Devices

Riccardo Spolaor[1(✉)], Laila Abudahi[2], Veelasha Moonsamy[3], Mauro Conti[1], and Radha Poovendran[2]

[1] University of Padua, Padua, Italy
{rspolaor,conti}@math.unipd.it
[2] University of Washington, Seattle, USA
{abudahil,rp3}@uw.edu
[3] Radboud University, Nijmegen, The Netherlands
email@veelasha.org

**Abstract.** More and more people are regularly using mobile and battery-powered handsets, such as smartphones and tablets. At the same time, thanks to the technological innovation and to the high user demand, those devices are integrating extensive battery-draining functionalities, which results in a surge of energy consumption of these devices. This scenario leads many people to often look for opportunities to charge their devices at public charging stations: the presence of such stations is already prominent around public areas such as hotels, shopping malls, airports, gyms and museums, and is expected to significantly grow in the future. While most of the times the power comes for free, there is no guarantee that the charging station is not maliciously controlled by an adversary, with the intention to exfiltrate data from the devices that are connected to it.

In this paper, we illustrate for the first time how an adversary could leverage a maliciously controlled charging station to exfiltrate data from the smartphone via a USB charging cable (i.e., without using the data transfer functionality), controlling a simple app running on the device—and without requiring any permission to be granted by the user to send data out of the device. We show the feasibility of the proposed attack through a prototype implementation in Android, which is able to send out potentially sensitive information, such as `IMEI` and contacts' phone number.

## 1 Introduction

Market studies predicted that in 2011 smartphone sales would surpassed that of desktop PCs [31]. To this date, smartphones remain the most used handheld devices. This is partly due to the fact that these devices are more powerful and provide more functionalities than the traditional feature phones. As a result, users can perform a variety of tasks on an actual smartphone device, which in the past would have been possible only on a desktop PC. In order to carry out such tasks, the smartphone platform offers its users a plethora of applications (apps).


D. Gollmann et al. (Eds.): ACNS 2017, LNCS 10355, pp. 83–102, 2017.
DOI: 10.1007/978-3-319-61204-1_5

Moreover, as users are constantly using apps (e.g., the gaming app, Pokémon Go) and would eventually require to recharge their smartphones, the demand for public charging stations have increased significantly in the last decade. Such stations can be seen in public areas such as airports, shopping malls, gyms and museums, where users can recharge their devices for free. In fact, this trend is also giving rise to a special type of business[1], which allows shop owners to install charging stations in their stores so as to boost their sales by providing free phone recharge to shoppers.

As the phone recharging is usually for free, however, at the same time one cannot be sure that the public charging stations are not maliciously controlled by an adversary. The Snowden revelations gave us proof that civilians are constantly under surveillance and nations are competing against each other by deploying smart technologies for collecting sensitive information en mass. In our work, we consider an adversary (e.g., manufacturers of public charging stations, Government agencies) whose aim is to take control over the public charging station and whose motive is to exfiltrate data from the user's smartphone once the device is plugged into the station.

In this paper, we demonstrate the feasibility of using power consumption (in the form of power bursts) to send out data over a Universal Serial Bus (USB) charging cable, which acts as a covert channel, to the public charging station. We implemented a proof-of-concept app, *PowerSnitch*, that can send out bits of data in the form of power bursts by manipulating the power consumption of the device's CPU. Interestingly, PowerSnitch does not require any special permission from the user at install-time (nor at run-time) to exfiltrate data out of the smartphone over the USB cable. On the adversary's side, we designed and implemented a decoder to retrieve the bits that have been transmitted via power bursts. Our empirical results show that we can successfully decode a payload of 512 bits with a 0% Bit Error Ratio (BER). In addition, we stress that the goal of this paper is to assess for the first time the feasibility of data transmission on such a covert channel and not to optimize its performance, which we will tackle as future work.

We focus primarily on Android, as it is currently the leading platform and has a large user base. However, we believe that this attack can be deployed on any other smartphone operating systems, as long as the device is connected to a power source at the public charging station.

Our contributions are as follows:

1. To our knowledge, we are the first to demonstrate the practicality of using the power feature of a USB charging cable as a covert channel to exfiltrate data, in the form of power bursts, from a device while it is connected to a power supplier. The attack works in Airplane mode as well.
2. We implemented a prototype of the attack, i.e., we designed and implemented its two components: (i) We built a proof-of-concept app, *PowerSnitch*, which does not require any permission granted by the user to communicate bits

[1] chargeitspot.com, chargetech.com.

of information in the form of power bursts back to the adversary; (ii) The decoder is deployed on the adversary side, i.e., public charging station to retrieve the binary information embedded in the power bursts.
3. We are able with our prototype to actually send out data using power bursts. Our prototype demonstrate the practical feasibility of the attack.

The rest of the paper is organized as follows. In Sect. 2, we present a brief literature overview of covert channel and data exfiltration techniques on smartphones. In Sect. 3, we include some background knowledge on Android operating system, and signal transmission and processing. In Sect. 4, we provide a description of our covert channel and decoder design, followed by the experimental results in Sect. 5 and discussion in Sect. 6. We conclude the paper in Sect. 7.

## 2 Related Work

In this section, we survey the existing work in the area of covert channels on mobile devices. We also present other non-conventional attack vectors, such as side channel information leakage via embedded sensors which can be used for data exfiltration.

*Covert Channels* – A covert channel can be considered as a secret channel used to exfiltrate information from a secured environment in an undetected manner. Chandra et al. [8] investigated the existence of different covert channels that can be used to communicate between two malicious applications. They examined the common resources (such as battery) shared between two malicious applications and how they could be exploited for covert communication. Similar studies presented in [14,18,21,26] exploited unknown covert channels in malicious and clean applications to leak out private information.

As demonstrated by Aloraini et al. [1], the adversary is further empowered as smartphones continue to have more computational power and extensive functionalities. The authors empirically showed that speech-like data can be sent over a cellular voice channel. The attack was successfully carried out with the help of a custom-built rootkit installed on Android devices. In [10], Do et al. demonstrated the feasibility of covertly exfiltrating data via SMS and inaudible audio transmission, without the user's knowledge, to other mobile devices including laptops.

In our work, we present a novel covert channel which exploits the USB charging cable by leaking information from a smartphone via power bursts. Our proposed method is non-invasive and can be deployed on non-rooted Android devices. We explain the attack in more detail in Sect. 4.1.

*Power Consumption by Smartphones* – In order to prolong the longevity of the smartphone's battery, it is crucial to understand how apps consume energy during execution and how to optimize such consumption. To this end, several works [4,6,23,33] have been proposed. Furthermore, the authors from [13,17]

studied apps' power consumption to detect anomalous behavior on smartphones, thus leading to detection of malware.

Since existing work focus on energy consumption on the device, our attack would therefore go undetected as the smartphone's CPU sends small chunks of encoded data, which are translated into power bursts, back to the public charging station. Additionally, state-of-the-art attacks that have been performed while the smartphone is charging [15,19] exploit vulnerabilities of USB interface rather than actual energy consumption.

*Attack Vectors using Side Channel Leaks* – Modern smartphones are embedded with a plethora of sensors that allow users to interact seamlessly with the apps on their smartphones. However, these sensors have access to an abundance of information stored on the device that can get exfiltrated. These data leaks can be used as a side channel to infer, otherwise undisclosed, sensitive information about the user or device [2,16,32].

The authors from [3,22] demonstrated how accelerometer readings can be used to infer tap-, gesture- and keyboard-based input from users to unlock their smartphones. Similarly, Spreitzer [27] showed that the ambient-light sensor can be exploited to infer users' PIN input. Moreover, considering network traffic as a side-channel, it is possible to identify the set of apps installed on a victim's mobile device [28,29], and even infer the actions the victim is performing with a specific app [9].

As pointed out in the aforementioned existing work, the adversarial model did not require any special privileges to exploit side channel leaks to recover data exfiltrated via sensors. In this paper, we show that our custom app, PowerSnitch, does not require any special permissions to be granted by the user in order to communicate information (in terms of power bursts) to the adversary. Furthermore, we stress that while the `INTERNET` permission is one approach of data exfiltration, our proposed work is different as we show the feasibility and practicability of using a USB cable to exfiltrate data. In particular, our attack still works even when the phone is switched to Airplane mode and defeats existing USB charging protection dongles, as in [7], since we only require the USB power pins to exfiltrate data.

## 3 Background Knowledge

In this section, we briefly recall several concepts that we use in our paper about Android operating system in Sect. 3.1, and signal transmission and processing in Sect. 3.2.

### 3.1 Android System and Permissions

In the Android Operating System (OS), apps are distributed as `APK` files. These files are simple archives which contain bytecode, resources and metadata. A user can install or uninstall an app (thus the APK file) by directly interacting with

the smartphone. When an Android app is running, its code is executed in a sandbox. In practice, an app runs isolated from the rest of the system, and it cannot directly access other apps' memory. The only way an app could gain memory access is via the mediation of inter-process communication techniques made available by Android. These measures are in place to prevent the access of malicious apps to other apps' data, which could potentially be privacy-sensitive.

Since Android apps run in a sandbox, they not only have restriction in shared memory usage, but also to most system resources. Instead, the Android OS provides an extensive set of Accessible Programming Interfaces (APIs), which allows access to system resources and services. In particular, the APIs that give access to potentially privacy-violating services (e.g., camera, microphone) or sensitive data (e.g., contacts) are protected by the Android Permission System [11]. An app that wants access to protected data or service must declare in the form of permission (identified by a string) in its manifest file. The list of permissions needed by an app is shown to the user when installing the app, and cannot be changed while an app is installed on the device. With the introduction of Android M (i.e., 6.0), permissions can be dynamically granted (by users) during an app's execution.

The permission system has also the goal of reducing the damage in case of a successful attack that manages to take control of an app, by limiting the resources that app's process has access to. Unfortunately, permission over-provisioning is a common malpractice, so much so that research efforts have been spent in trying to detect this problem [5]. Moreover, an app asking for permissions not related to its purpose (or functionality) can hide malicious behaviors (i.e., spyware or malware apps) [20].

### 3.2 Signal Transmission and Processing

In this section, we provide some background information on bit transmission, and signal processing and decoding used in our proposed decoder (see Sect. 4.4).

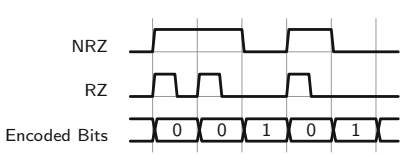
*Bit Transmission* – To enable bit transmission over our channel, an understanding of basic digital communication systems is essential. For proof-of-concept purposes, the design of our bit transmission system was inspired by amplitude-based modulation in the digital communication literature.

Amplitude-Shift Keying (ASK) is a form of digital modulation where digital bits are represented by variations in the amplitude of a carrier signal. To send bits over our channel, we used On-Off Signaling (OOS), which is the simplest form of ASK where digital data is represented by the presence and absence of some pulse *p(t)* for a specific period of time. Figure 1a shows the difference between a Return-to-Zero (RZ) and a Non-Return-to-Zero (NRZ) on-off encoding. In NRZ encoding, bits are represented by a sufficient condition (a pulse) that occupies the entire bit period $T_b$ while RZ encoding represents bits as pulses for a duration of $T_b/2$ before it returns to zero for the following $T_b/2$ period.

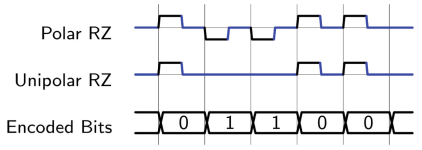
On the other hand, Fig. 1b shows the difference between a unipolar and a polar RZ on-off signaling. In a polar RZ encoding, two different conditions,

different-sign pulses are used to encode different bits(zeros/ones) while the presence and absence of a single pulse, a positive one in our case, are used to encode different bits.

For the sake of our channel design, it is safe to assume that we can only increase the power consumption of a phone at certain times and hence, are able to generate only positive (high) bursts. Thus, a unipolar encoding seems more relative and applicable for our channel. Moreover, successive peaks, such as the first two zeros in Fig. 1a, are easier to identify, and thus decode, in the RZ-encoded signal than in the NRZ one. This advantage of RZ over NRZ becomes especially apparent in cases where the bit period is expected not to be restrictively fixed in the received signal whether it is due to expected high channel noises or lack of full control of the phone's CPU. Therefore, unipolar RZ on-off signaling was used to encode leaked bits over our covert channel.




(a) Return-to-Zero (RZ) and Non-Return-to-Zero (NRZ) On-Off Encoding.




(b) A Polar and a Unipolar encoding of an RZ On-Off Signal.

**Fig. 1.** A comparison between bit encoding methods

*Signal Processing and Decoding* – After choosing the appropriate encoding method to transmit bits, it is also essential to think about the optimal receiver design and how to process the received signal and decode bits with minimum error probability at the receiver side of the channel. As known in the digital communication literature, matched filters are the optimal receivers for Additive White Gaussian Noise (AWGN) channels. We refer the reader to Sect. 4.2 of [24] for a detailed proof.

Matched Filters are obtained by correlating the received signal $R(t)$ with the known pulse that was first used to encode a transmitted bit, in this case P(t) with period $T_b$. After correlation, the resulted signal is then sampled at time $T_b$, which means that the sampling rate equals to $1/T_b$ samples/seconds. This way, each bit is guaranteed to be represented by only one sample. The decoding decision will then be made based on that one sample value; if the sample value is more than a given threshold, this indicates the presence of P(t); and hence a zero in our case, while a sample value below the threshold indicates the absence of P(t) and hence a one is decoded.

However and most importantly, for matched filters to work as expected, it is essential to have fixed bit period $T_b$ throughout the entire received signal. If the periods of the received bits were varying, the matched filter samples taken with the $1/T_b$ sampling rate will not be as optimal and representative of the bit data as expected and synchronization will be lost.

Since there exist infrequent phone-specific, OS-enforced conditions that can affect the power consumption of a phone, the noises on our channel are expected to be more complex to fit in an AWGN model. Hence, a matched filter receiver is most likely not the optimal receiver for our channel. More creative decoder design decisions are needed to maximize the throughput of our channel and minimize the error probability.

## 4 Covert Channel Using Mobile Device Energy Consumption

In this section, we elaborate on the components that make up our covert channel attack. We begin by giving an overview of the attack in Sect. 4.1. We then define the terms and parameters for transmission in Sect. 4.2, followed by a description of each component of the attack: PowerSnitch app in Sect. 4.3 and the energy traces decoder in Sect. 4.4.

### 4.1 Overview of Attack

As illustrated in Fig. 2, the attack scenario considers two components: the victim's Android mobile device (sender) and an accomplice's power supplier (receiver). Victim's mobile device is connected to a power supplier (controlled by the adversary) through a USB cable.

The left side of Fig. 2 depicts what happens after the victim has installed our proof-of-concept app, *PowerSnitch*. The app is able to exfiltrate victim's private information, which gets encoded as CPU bursts with a specific timing. Indeed, as the CPU is one of the most energy consuming resources in a device, a CPU burst can be directly measured as a "peak" based on the amount of energy absorbed by a mobile device. The right side of Fig. 2 illustrates how the energy supplier is able to measure (with a given sampling rate) the electric current provided to the mobile device connected to the public charging station. Then, such electric measurement, which is considered as a *signal*, is given as input to a decoder. It should be noted that the adversary, i.e., the public charging station, has control of the power supplier, and thus is able to control the amount of current provided to the device – even if it has the "fast charge" capability.

In our proposed covert channel attack, we consider situations in which users connect their mobile devices for more than 20 min. There are several scenarios that fulfill such time requirements. Examples are: (i) recharging a device overnight in a hotel room; (ii) making use of locked boxes in shopping malls for charging mobile phones; (iii) recharging devices on planes, in trains and cars.

In addition, we argue that those time requirements are more than reasonable since generally, 72% of users leave their phones on charging for more than 30 min, with an average time of 3 h and 54 min, as reported in [12]. This means that: (i) the mobile device is in stand-by mode; (ii) CPU and the use of other energy consuming resources (e.g., Wi-Fi or 3/4g data connection) usage is limited only to the OS and background apps. Moreover, since there is no user interaction, it

is reasonable to assume that the phone screen, which has a relevant impact on energy consumption, will stay off for the aforementioned period of time.

Moreover, it is also worth noting that the attack is still feasible if there is no data connection between the victim's device and the power supplier, such as Media Transfer Protocol (MTP), Photo Transfer Protocol (PTP), Musical Instrument Digital Interface (MIDI). This is possible as our methodology only requires power consumption to send out the power bursts. Moreover, from Android version 6.0, when a device is connected via USB, it is set by default to "Charging" mode (i.e., just charge the device), thus no data connection is allowed unless the user switches on data connection manually. This improvement in security feature does not impact our proposed attack as we do not make use of data connection to transfer the power bursts.

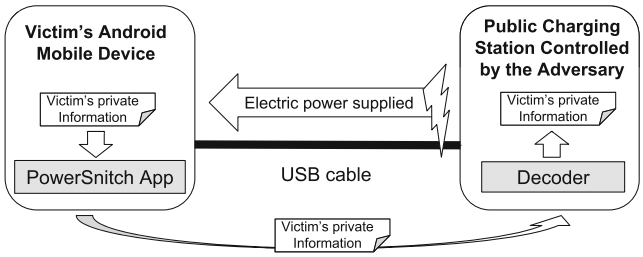



**Fig. 2.** The schema of the components involved in the attack.

### 4.2 Terminology and Transmission Parameters

In this section, we define the necessary terminology to identify concepts used in the rest of the paper:

- *Payload* is the information that has to be sent from the device to the receiver.
- *Transmission* is the whole sequence of bits transmitted in which the payload is encoded.

In order to obtain a successful communication, the sender and the receiver need to agree on the parameters of the transmission.

- *Period* is the time interval during which a bit is transmitted.
- *Duty cycle* is the ratio between burst and rest time in a period $T_b$. For example: if a burst lasts for $T_b/2$, the duty cycle will be 50%.
- *Preamble* is the sequence of bit used to synchronize the transmission. Usually a preamble is used at the beginning of a transmission, but it can also be used within a transmission in order to recover the synchronization in case of error. In our case, we used a preamble composed of 8 bits.

### 4.3 PowerSnitch App: Implementing the Attack on Android

The first component of our covert channel we discuss is the proof-of-concept which we called *PowerSnitch app*. This app, used for the covert channel exploit, has been designed as a service in order to be installed as a standalone app or a library in a repackaged app. Henceforth, we refer to both these variants simply with the term "app".

PowerSnitch app requires only the `WAKE_LOCK` permission and does not need root access to work. Such permission allows PowerSnitch app to wake and force execute the CPU while the device is in sleep mode, so that it can start to transmit the payload. We stress that since it is running as a background service, PowerSnitch app still works even when user authentication mechanisms (e.g., PIN, password) are in place. Moreover, since it does not use any conventional communication technology (e.g., Wi-Fi, Bluetooth, NFC), PowerSnitch app can exfiltrate information even if the device is in airplane mode. It is worth mentioning that Android M (i.e., 6.0) introduced the Doze mode [30], a battery power-saving optimization which reduces the apps activity when the device is inactive and running on battery for extended periods of time. When it is in place, Doze mode stops background CPU and network activity (ignoring wakelocks, job scheduler, Wi-Fi scan, etc.). Then on periodic time intervals (i.e., maintenance windows), the system runs all pending jobs, synchronization and alarms. However, such optimization is not active when a device is connected to a power source or when the screen is on. This means that Doze does not affect our proposal since we need the wakelock function but also the device to be plugged to a power source. Moreover, since our proposed attack needs also the status of the battery, it does not need any permission in order to obtain such information: in fact, it is sufficient to only register at run-time (not even in the manifest) a specific broadcast receiver (i.e., `ACTION_BATTERY_CHANGED`).

In Fig. 3, we illustrate the modules of PowerSnitch app. It is composed of three modules: *Payload encoder*, *Transmission controller* and *Bursts generator*. *Payload encoder* takes the payload as input and outputs an array of bits. The payload can be any element that can be serialized into an array of bits. We use strings as payloads, they are first decomposed into an array of characters and then, using the ASCII code of each character, into an array of bits. *Payload Encoder* can also add to its output array synchronization bits (e.g., the preamble), and error checking codes (e.g., CRC).

*Transmission controller* is in charge of monitoring the status of the device with the purpose of understanding when it is feasible to transmit through the covert channel. Indeed, in order to not be detected by the user, it checks whether all the following conditions are satisfied: (i) the USB cable is connected; (ii) the screen is off; and (iii) the battery is sufficiently charged (see Sect. 6). If our app receives a broadcast intent from the Android OS that invalidates one of the aforementioned conditions, *Transmission controller* module will interrupt the transmission. It is worth noticing that to obtain all this information, PowerSnitch app does not need any additional permission. From the GUI app used in

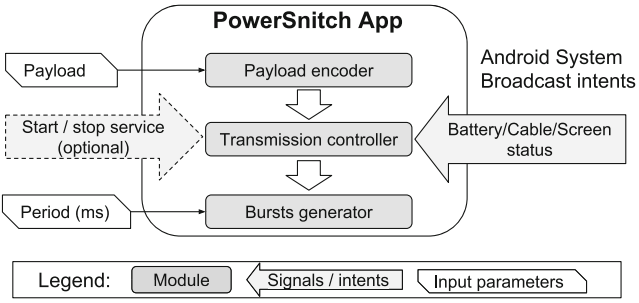



**Fig. 3.** The modules involved in the PowerSnitch app.

our experiments, we are also able to start or stop PowerSnitch app (represented in Fig. 3 with a dotted arrow).

The last component of PowerSnitch app is *Bursts generator*. The task of this component is to convert the encoded payload into bursts of energy consumption. These bursts will generate a signal that can be measured at the other end of the USB cable (i.e., the power supplier). In order to obtain these bursts of energy consumption, *Bursts generator* module can use a power consuming resource of the mobile device such as CPU, screen or flashlight. Our proof-of-concept, *Bursts generator* uses the CPU: a CPU burst is generated from a simple floating point operation repeated in a loop for a precise amount of time (given by transmission parameters).

### 4.4 Analysis of Energy Traces

To make better decoder design decisions, several channel traces were observed, collected and then used to calculate channel estimations and implement different simulations of the channel performance and behavior. A standard on-off signaling decoder needs to know the exact period of bits in the received signal in order to be able to decode them. However, a channel built based on a phone's power consumption is expected to have hard-to-model noises that, after examining the collected channel data traces, are actually affecting not only the peak periods but also the peak amplitudes. The amount of external power consumed by a phone can be largely affected by dominant OS-enforced, manufacturer-specific factors. For instance, different sudden drop patterns in power consumption especially when the phone is almost or completely charged, lack of control over the OS scheduler; when, how often and for how long do some heavy power-consuming OS background services run, as well as the precision and sampling rate of the power monitor on the receiver side of the channel.

Figure 4 shows a portion of the channel data captured after a transmission of ten successive bits (ten Zeros, therefore ten peaks) was initiated by our app on a Nexus 6 phone. It should be noted that the data was passed through a low-pass filter to get rid of harsh, high frequency noises in order to make the signal looks smoother. As a result, based on a threshold of 100 mA, ten peaks are

successfully detected. Moreover, the width of each peak, and hence the period of each bit, is varying sufficiently. The first bit, for example, has a period of 300 ms while the eighth one has a period of only 195 ms. Although the intended bit period generated and transmitted by the app was 500 ms, the average period of the received bits was actually 311 ms, which the receiver has no way to predict in advance. Such variations in the received signal are expected to affect the performance of any decoder. An ideal matched filter receiver will have hard time decoding such inconsistent signal and synchronization will be lost very quickly. We elaborate further on this issue in the remaining sections.

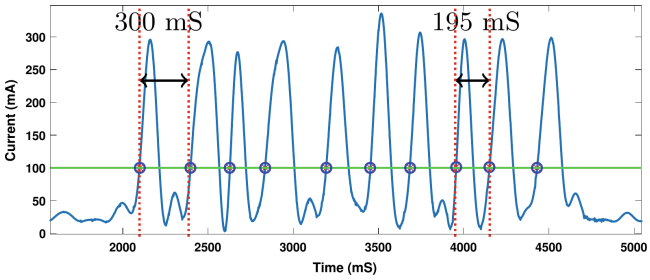



**Fig. 4.** A portion of a received signal showing the variations in peak widths and amplitudes.

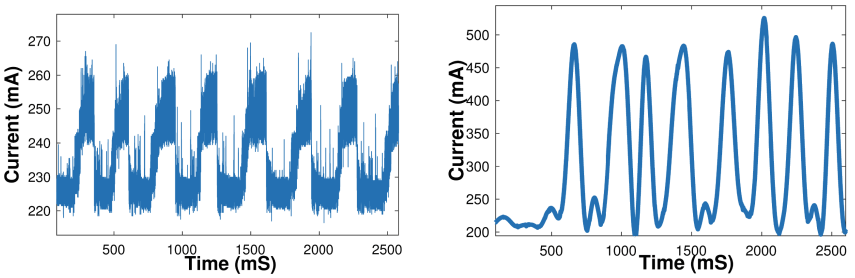
**Decoder Design.** In this section, we provide additional explanation about the different processing stages that our decoder is taking the received signal through in order to overcome the channel inconsistencies and decode the sent bits with the minimum Bit Error Ratio (BER). In signal processing, the quality of a communication channel can be measured in terms of BER (represented as a percentage), which is the number of bit errors divided by the total number of transmitted bits over the channel. Channels affected by interference, distortion, noise, or synchronization errors have a high BER.

Figure 5 summarizes the different processing stages which will be discussed in the order they take place in, along with some background information and algorithm justifications, where applicable.


**Fig. 5.** Different phases of our decoder.

*Data Filtering.* First, the received signal is passed through a low-pass filter to get rid of the harsh high-frequency noises. For instance, Fig. 6 shows the same portion of a received signal before and after applying the low-pass filter. The low-pass filter helps not only to make the signal looks smoother, but also to

make the threshold-based detection of real peaks easier by eliminating narrow-peak noises that can be falsely identified as real peaks or bits. Additionally, the low-pass filter used in our decoder adjusts its pass and stop frequencies based on the intended bit period generated by the phone in order to make sure that we do not over-filter or over-attenuate the signal.




(a) Raw received signal. (b) Low-pass filtered received signal.

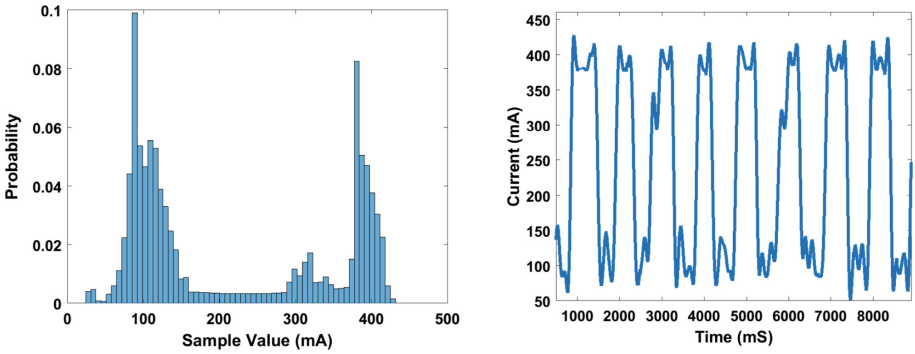
**Fig. 6.** A portion of a received signal before and after applying the low-pass filter.

*Threshold Estimation.* The decoder detects peaks by decoding unipolar RZ on-off encoded bits. The presence or absence of a peak (a 0 or a 1 in our case, respectively) at a certain time and for a specific period is then translated to the corresponding bit. Peak detection is usually done by setting an appropriate threshold; anything above the threshold is a peak and anything below is just noise. However, deciding which threshold to use is not a trivial process especially with the unpredictable noise in our channel and the variations in width and amplitude of the received peaks.

The threshold value used by the decoder is highly critical to peaks detection, the resulted width of detected peaks and the decoder performance. Hence, we primarily use a known preamble data sent prior to the actual packet to estimate the threshold. The preamble consists of eight known bits (eight zeros in our case) at the start of the transmission, which means that the decoder is expecting eight peaks at the start. Since a unipolar RZ on-off encoded zero has a pulse for half of the bit period, the preamble is expected to have roughly the same number of peak and no-peak samples. Therefore, a histogram of the preamble samples is expected to split into two portions; peak and no-peak portions. Figure 7a shows a histogram of the preamble samples shown in Fig. 7b. As observed, the histogram has two distinguishable densities; each of them look like the probability density function of a Gaussian distribution.

Estimating the parameters (mean and variance) of two Gaussians that are believed to exist in one overall distribution is a complicated statistical problem. However, the Gaussian Mixture Model (GMM), introduced and explained in [25], is a probabilistic model commonly used to address this type of problem and

to statistically estimate the parameters of existing Gaussian populations. To estimate the threshold, as shown in Fig. 8, the decoder uses the GMM to fit two Gaussians to the two histogram portions, find the mean of each one of them and then compute the threshold as the middle point between the two means. As a result, our decoder is able to estimate the threshold independently and without any previous knowledge of the expected amplitudes of the received bits. After that, each sample is converted to either a peak sample or no-peak sample based on whether the sample value is above or below the estimated threshold.




(a) A histogram of the preamble samples. (b) A received preamble signal.

**Fig. 7.** A histogram of the preamble samples shows a mixture of two Gaussian-like densities.

*Robust Decoding.* Generally, the way a decoder translates the peak and no-peak samples to zeros and ones is highly time-sensitive. For instance, if the bit period is fixed and equals to $T_b$, the decoder simply checks the presence or absence of the peak in each $T_b$ period. Since this decoding decision is made based on a very strict timing manner, the slightest error in the received bit periods will cause a quick loss of synchronization. As mentioned in the previous section, the received peak widths (and hence bit periods) over our channel are changing with a high variation around their mean. Therefore, our decoding decision cannot rely on an accurate notion of time. Instead, our decoder needs to assume a sufficient amount of error in the period of each received bit and to search for the peaks in a wider range instead of a strict period of time.

To address this level of time-insensitivity and achieve robustness to synchronization errors, our decoding decision was made based on the time difference between each two successive peaks. As an example, assume that two successive zeros were sent and hence two peaks were received. The difference between the start time of each peak should be rounded to the average bit period. It should be noted that the decoder computes the average bit period based on the received preamble data. However, if a zero-one-zero transmission was made, the time difference of the start of the two received peaks should be rounded to double of

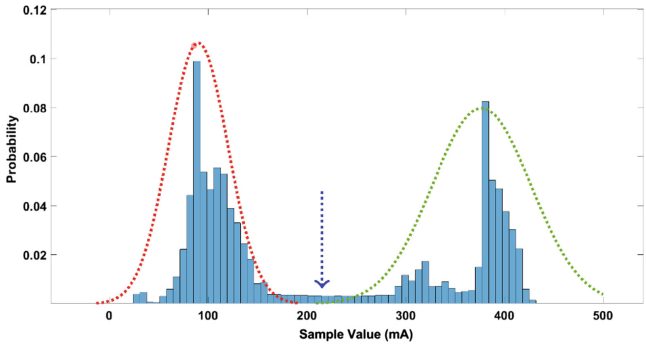



**Fig. 8.** Using the Gaussian Mixture Model to estimate the threshold.

the average bit period. If a zero-one-one-zero transmission was made, the difference should be rounded to triple the average period and so on. Eventually, synchronization is regained with every detected peak and based only on the time difference between peaks, the decoder makes a decision on how many no-peak bits (ones in our case) are transmitted between the zeros. The time difference does not have to be exactly equal to a multiple of the average bit period. Instead, a range of values can be rounded up to the same value and thus more flexible time-insensitive decoding decision is made.

## 5 Experimental Evaluation

In this section, we first describe the devices used in our experiments and the values for transmission parameters. We then report the results of the transmission evaluation.

### 5.1 Experiment Settings

In our experiments, we programmed the PowerSnitch app using Android Studio with API. The device used to measure the energy provided to the device via USB cable is Monsoon Power Monitor[2] in USB mode with 4.55 V in output. The decoder used to process signal was implemented in MATLAB. In order to evaluate the performance of the transmission, we send out a payload comprised of letters and numbers of ASCII code for a total of 512 bits. The values of period used range from 500 ms to 1000 ms with increments of 100 ms. It is worth mentioning that bits sent over our channel were not packeted and no error detection or correction techniques were used. For each phone and bit period, BER was computed after sending 512 bits at once and then number of bits that were incorrectly decoded was calculated.

[2] www.msoon.com/LabEquipment/PowerMonitor.

We evaluate the performance of our proposal on the following devices running Android OS: Nexus 4 with Android 5.1.1 (API 22), Nexus 5 with Android 6.0 (API 23), Nexus 6 with Android 6.0 (API 23) and Samsung S5 with Android 5.1.1 (API 22). We underline that the devices used in our experiments are actual personal devices, kindly lent by some users without any money reward. In order to replicate an actual real world scenario, we did not uninstall any app, nor stopped any app running in background. The only intervention we made on those devices is the installation of our PowerSnitch app.

### 5.2 Results

In Table 1, we report the performance of the decoder for processing the received power bursts on different mobile devices. The results presented in the table are in terms of BER in the transmission of the payload; the lower the BER, the better is the quality of the transmission. For Nexus devices (i.e., Nexus 4, 5 and 6), we achieve a zero or low BER of periods of 800 ms and 900 ms (i.e., 1.25 and 1.11 bits per seconds, respectively). While for Nexus 4 and 6, the BER remains under 20% and, for Nexus 5, it increases to 37% and 40% with periods 700 ms and 600 ms, respectively. For Samsung S5, the transmission BER is at 12.5% with a period of 1 s, and it slowly increases to around 21% with a period of half a second.

The higher BER for Nexus 5 (i.e., periods 700 ms and 600 ms in Table 1) are due to de-synchronization of the signal that the decoder was not able to recover. To cope with this problem, we can divide the payload into packets, where a packet header will be the preamble in order to recover the synchronization. A quick overview of the communication literature can show how a BER of 30% can be recovered using a simple Forward Error Correction (FEC) technique where the transmitter encodes the data using an Error Correction Code (ECC) prior to transmission; for example bits redundancy or parity checks.

**Table 1.** Results in terms of Bit Error Ratio (BER) as percentage.

| Device | | Period (milliseconds) | | | | | |
|---|---|---|---|---|---|---|---|
| Model | Operating system version | 1000 | 900 | 800 | 700 | 600 | 500 |
| Samsung S5 | Android 5.1.1 (API 22) | 12.5 | 13.5 | 13.31 | 16.33 | 17.9 | 21.42 |
| Nexus 4 | Android 5.1.1 (API 22) | 13.5 | 0.78 | 0.0 | 0.0 | 13.33 | 16.21 |
| Nexus 5 | Android 6.0 (API 23) | 21.0 | 0.0 | 0.95 | 36.82 | 40.35 | 13.4 |
| Nexus 6 | Android 6.0 (API 23) | 1.07 | 0.0 | 0.21 | 0.0 | 4.05 | 7.42 |

## 6 Discussion and Optimizations

In this section, we elaborate further on the results obtained in the experimental evaluation of our proposed attack (Sect. 5). In particular, we discuss on interesting

observation made during our experiments. We also present the optimizations that were implemented in the framework in order to make our proposed attack more robust.

An interesting phenomenon to notice is that, as observed in our experiments, the level of battery affects the quality of the transmission signal. In Fig. 9, we present the amount of electric current provided by the power supplier to a Nexus 6 during recharge (i.e., the first 35 min) and full battery states (i.e., after 35 min). Indeed, when the level for the battery is low (i.e., 0% to around 40%) the device consumes a high amount of energy, and almost all of it is used to recharge the battery.

When attempting to transmit data in the aforementioned conditions, we discover that the bursts were not easily distinguishable. In fact, the difference in terms of energy consumption between burst and rest was so small that it cannot be distinguished from noise; thus, they can be filtered out during the signal processing. Additionally, when the level of the battery is increased, the amount of energy consumed to recharge the battery gradually decreases. We observed that when the battery level is higher than 50%, the power bursts become more and more distinguishable. However the best condition under which the bursts are clear is when the battery is fully charged. Indeed, as we can notice from Fig. 9, the current drops down after the battery level reaches 100%, because there is no need to provide energy to the battery anymore - except to keep the device running.

The percentages mentioned above also depends from the power supplier used to provide energy to the device. In our experiments, we used Monsoon power monitor which provides as output at most 4.55 V. Due to the limitation of such power monitor, during the recharge of devices with fast charge technology (e.g., Samsung S5, Nexus 6 and 6P), which are able to work with 5.3 V and 2 mA, the energy consumed is almost constant until the battery is almost fully charged. Thus, we cannot decode any signal from the energy consumption.

In order to avoid to transmit when the receiver is not able to decode the signal, PowerSnitch checks whether the battery level is among a certain threshold $\omega$. Such threshold $\omega$ can be obtained by PowerSnitch itself, simply knowing the model in which it is running. This information can be easily obtained without any permission (`android.os.Build.MODEL` and `MANUFACTURER`).

*Optimizations.* In what follows, we elaborate on the optimizations that were implemented in order to not be detected or make the victim suspicious. The first optimization is to keep a duty cycle (i.e., the time of burst in a period) under 50%. During an attack, if such optimization is not taken into account (i.e., a duty cycle greater than 75%), the victim may be alerted by two possible effects:

- the temperature of the device could increase significantly, in a way that could be perceived by touching it.
- if the attack takes place during the battery charge phase, the battery will take more time to recharge due to the high amount of energy used by CPU.

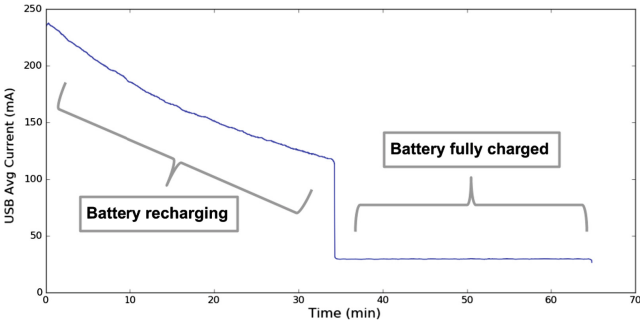



**Fig. 9.** Electric current provided to a Nexus 6 during recharge phase and battery fully charged.

However, as previously explained in Sect. 3.2, the duty cycle should be 50% of period (i.e., $T_b/2$) in order to achieve a RZ. Thus, the above effects are already taken care of in our proposed attack.

Another optimization involves the Android Debug Bridge (ADB) tool. It is possible to monitor CPU consumption of an Android device via ADB. Hence, one may use such debug tool to detect that something strange is happening on the device (i.e., a transmission on the covert channel using CPU bursts). Fortunately, PowerSnitch app could easily detect whether ADB setting is active through `Settings.Global.ADB_ENABLED`, once again provided by an Android API.

Another optimization to PowerSnitch app would be the ability to detect if the power supplier is an accomplice of the attack. The accomplice has to let PowerSnitch app know that it is listening to the covert channel by communicating something equivalent to a "hello message". In order to do so, we can rely on the information about the amount of electric current provided to recharge the battery. Such information is made available through `BatteryManager` object, provided by Android API. In particular, `BATTERY_PROPERTY_CURRENT_NOW` data field (available from API 21 and on devices with power gauge, such as Nexus series) of `BatteryManager` records an integer that represents the current entering the battery in terms of mA.

On one hand, the power supplier can then variate the current in output above and below a certain threshold $\theta$ with a precise timing. As a practical and non-limiting example, at a point in time during the recharging, the power supplier can output current with the following behavior: (i) below $\theta$ for $t$ seconds, (ii) above $\theta$ for $t$ seconds, (iii) again below $\theta$ for $t$ seconds and finally (iv) above $\theta$ for good. On the other hand, since PowerSnitch app monitors `BATTERY_PROPERTY_CURRENT_NOW` and knows the aforementioned behavior (along with both $\theta$ and $t$), it will be able to understand that at the other end of the USB cable there is an accomplice power supplier ready to receive a transmission. This optimization is significant for reducing the chance to remain undetected, since PowerSnitch app will transmit data if and only if it is sure that an accomplice power supplier is listening. With such optimization, we will obtain a half-duplex communication channel, since the

communication is bidirectional but only one participant (i.e., the device or the power source) is allowed to transmit at a time. This optimization is not currently implemented and will be considered as future work.

To summarize, the conditions under which the transmission of data is optimal and the chance of being detected is lowest are as follows: the mobile device has to be charged more than 50%, the screen has to be off, ADB tool should be switched off (which is true by default) and the phone must to be plugged with a USB charging cable to a public charging station which is controlled by the adversary.

## 7 Conclusion

In this paper, we demonstrate for the first time the practicality of using a (power-only) USB charging cable as a covert channel to exfiltrate data from a smartphone, which is connected to a charging station. Since there are no visible signs of the existence of a covert channel while the battery is recharging, the user is oblivious that data is being leaked from the device. Moreover, our proposed covert channel defeats existing USB charging protection dongles, as described in [7] because it requires only the USB power pins to exfiltrate data in the form of CPU power bursts.

To show the feasibility and practicality of our proposed covert channel, we implemented an app, *PowerSnitch*, which does not require the user to grant access to permissions at install-time (nor at run-time) on a non-rooted Android phone. Once the device is plugged in a compromised public charging station, the app encodes sensitive information and transmits it via power bursts back to the station. Our empirical results show that we are able to exfiltrate a payload encoded in power bursts at 1.25 bits per seconds with a BER under 1% on the Nexus 4-6 devices and a BER of around 13% for Samsung S5. As future work, we plan to investigate malicious power banks and how they can be exploited using our covert channel to exfiltrate data from smart devices. We will also work on the transmitter and decoder by extending the framework to include error correction algorithms and synchronization recover mechanisms to lower down the BER of data transmission—as this was not the main goal of this paper.

**Acknowledgments.** This work is supported by ONR grants N00014-14-1-0029 and N00014-16-1-2710, ARO grant W911NF-16-1-0485 and NSF grant CNS-1446866.

Veelasha Moonsamy is supported by the Technology Foundation STW (project 13499 - TYPHOON & ASPASIA) from the Dutch government.

Mauro Conti is supported by a Marie Curie Fellowship funded by the European Commission (agreement PCIG11-GA-2012-321980). This work is also partially supported by the EU TagItSmart! Project (agreement H2020-ICT30-2015-688061), the EU-India REACH Project (agreement ICI+/2014/342-896), "Physical-Layer Security for Wireless Communication", and "Content Centric Networking: Security and Privacy Issues" funded by the University of Padua. This work is partially supported by the grant n. 2017-166478 (3696) from Cisco University Research Program Fund and Silicon Valley Community Foundation.

We would like to thank Elia Dal Santo and Moreno Ambrosin for their insightful comments.

## References


1. Aloraini, B., Johnson, D., Stackpole, B., Mishra, S.: A new covert channel over cellular voice channel in smartphones. Technical report (2015). arXiv preprint arXiv:1504.05647
2. Aviv, A.J., Gibson, K., Mossop, E., Blaze, M., Smith, J.M.: Smudge attacks on smartphone touch screens. In: Proceedings of USENIX WOOT (2010)
3. Aviv, A.J., Sapp, B., Blaze, M., Smith, J.M.: Practicality of accelerometer side channels on smartphones. In: Proceedings of USENIX ACSAC (2012)
4. Baghel, S., Keshav, K., Manepalli, V.: An investigation into traffic analysis for diverse data applications on smartphones. In: Proceedings of NCC (2012)
5. Bartel, A., Klein, J., Le Traon, Y., Monperrus, M.: Automatically securing permission-based software by reducing the attack surface: an application to android. In: Proceedings of ACM ASE (2012)
6. Carroll, A., Heiser, G.: An analysis of power consumption in a smartphone. In: Proceedings of USENIX ATC (2010)
7. Chacos, B.: USB condom promises to protect your dongle from infected ports. PC World, August 2014. http://tinyurl.com/hvlqkrt
8. Chandra, S., Lin, Z., Kundu, A., Khan, L.: Towards a systematic study of the covert channel attacks in smartphones. In: Tian, J., Jing, J., Srivatsa, M. (eds.) SecureComm 2014. LNICSSITE, vol. 152, pp. 427–435. Springer, Cham (2015). doi:10.1007/978-3-319-23829-6_29
9. Conti, M., Mancini, L.V., Spolaor, R., Verde, N.V.: Analyzing android encrypted network traffic to identify user actions. IEEE TIFS **11**(1), 114–125 (2016)
10. Do, Q., Martini, B., Choo, K.K.R.: Exfiltrating data from android devices. Comput. Secur. **48**, 74–91 (2015)
11. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of ACM CCS (2011)
12. Ferreira, D., Dey, A.K., Kostakos, V.: Understanding human-smartphone concerns: a study of battery life. In: Proceedings of PerCom (2011)
13. Kim, H., Smith, J., Shin, K.G.: Detecting energy-greedy anomalies and mobile malware variants. In: Proceedings of ACM MobiSys (2008)
14. Lalande, J.-F., Wendzel, S.: Hiding privacy leaks in android applications using low-attention raising covert channels. In: Proceedings of ARES (2013)
15. Lau, B., Jang, Y., Song, C., Wang, T., Chung, P.H., Royal, P.: Mactans: injecting malware into IOS devices via malicious chargers. Black Hat, USA (2013)
16. Lin, L., Kasper, M., Güneysu, T., Paar, C., Burleson, W.: Trojan side-channels: lightweight hardware trojans through side-channel engineering. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 382–395. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04138-9_27
17. Liu, L., Yan, G., Zhang, X., Chen, S.: VirusMeter: preventing your cellphone from spies. In: Kirda, E., Jha, S., Balzarotti, D. (eds.) RAID 2009. LNCS, vol. 5758, pp. 244–264. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04342-0_13
18. Marforio, C., Ritzdorf, H., Francillon, A., Capkun, S.: Analysis of the communication between colluding applications on modern smartphones. In: Proceedings of USENIX ACSAC (2012)
19. Meng, W., Lee, W.H., Murali, S., Krishnan, S.: Charging me and i know your secrets!: towards juice filming attacks on smartphones. In: Proceedings of ACM CPS-SEC (2015)

20. Moonsamy, V., Rong, J., Liu, S.: Mining permission patterns for contrasting clean and malicious android applications. J. Future Gener. Comput. Syst. **36**, 122–132 (2013)
21. Novak, E., Tang, Y., Hao, Z., Li, Q., Zhang, Y.: Physical media covert channels on smart mobile devices. In: Proceedings of ACM UbiComp (2015)
22. Owusu, E., Han, J., Das, S., Perrig, A., Zhang, J.: ACCessory: password inference using accelerometers on smartphones. In: Proceedings of ACM HotMobile (2012)
23. Pathak, A., Charlie Hu, Y., Zhang, M.: Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with Eprof. In: Proceedings of ACM EuroSys (2012)
24. Proakis, J.G.: Intersymbol Interference in Digital Communication Systems. Wiley, Hoboken (2003)
25. Reynolds, D.: Gaussian mixture models. Encycl. Biom., 827–832 (2015)
26. Schlegel, R., Zhang, K., Zhou, X.Y., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: a stealthy and context-aware sound trojan for smartphones. In: Proceedings of NDSS (2011)
27. Spreitzer, R.: Pin skimming: exploiting the ambient-light sensor in mobile devices. In: Proceedings of ACM CCS SPSM (2014)
28. Stöber, T., Frank, M., Schmitt, J., Martinovic, I.: Who do you sync you are?: Smartphone fingerprinting via application behaviour. In: Proceedings of ACM WiSec (2013)
29. Taylor, V.F., Spolaor, R., Conti, M., Martinovic, I.: Appscanner: automatic fingerprinting of smartphone apps from encrypted network traffic. In: Proceedings of IEEE EuroS&P (2016)
30. Android Developers. Optimizing for Doze and App Standby. http://tinyurl.com/zvphw46
31. Business Insider. The Smartphone Market Is Now Bigger Than The PC Market (2011). http://goo.gl/XkM8XM
32. Yan, L., Guo, Y., Chen, X., Mei, H.: A study on power side channels on mobile devices. In: Proceedings of ACM Internetware (2015)
33. Yoon, C., Kim, D., Jung, W., Kang, C., Cha, H.: AppScope: application Energy metering framework for android smartphone using kernel activity monitoring. In: Proceedings of ATC (2012)