Inferring UI States of Mobile Applications through Power Side Channel Exploitation

Yao Guo, Junming Ma, Wenjun Wu, and Xiangqun Chen

Key Laboratory of High-Confidence Software Technologies (Ministry of Education),
School of EECS, Peking University, Beijing, China.
{yaoguo,majunming,wuwenjun,cherry}@pku.edu.cn

Abstract. The UI (user interface) state of a mobile application is important for attackers since it exposes what is happening inside an application. Attackers could initiate attacks timely according to this information, for example inserting fake GUIs or taking screenshots of GUIs involving user's sensitive data. This paper proposes PoWatt, a method to infer the timing of sensitive UI occurrences by exploiting power side channels on mobile devices such as smartphones. Based on power traces collected and power patterns learned in advance, PoWatt applies a pattern matching algorithm to detect target UI occurrences within a series of continuous power traces. Experiment results on popular Android apps show that PoWatt can detect sensitive UI loading with an average precision of 71%(up to 98%) and an average recall rate of 70%(up to 88%) during offline detection. In real-time experiments for online detection, PoWatt can still detect sensitive UIs with a reasonable precision and recall, which can be successfully exploited by real-world attacks such as screenshot-based password stealing. Finally, we discuss the limitations of PoWatt and possible mitigation techniques.

Key words: Side channels, power traces, power channels, UI inference, smartphones

1 Introduction

Side channel attacks have been studied extensively. The goal of side channel attacks is gaining confidential information from the targeted computing system, while leveraging *side channels* that are not directly revealing sensitive information. Previously discovered side channels include timing information [18, 4, 21], sound [22], shared memory/registers/files between processes [16] and power consumption [8, 15], etc.

Power side channels (or power analysis attacks) have become an important type of covert side channels. One well-known example of power side channels is the recovery of an encryption key from a cryptosystem [8, 7]. Messerges *et al.* examined both *simple power analysis(SPA)* and *differential power analysis (D-PA)* attacks [14] against the data encryption standard algorithm and managed to breach the security of smart-cards using signal-to-noise ratio (SNR) based multi-bit attack. For mobile systems such as smartphones, researchers have shown that power information can also be used to infer users' locations [15].

Goal Overview This paper introduces *a new power side channel*, which can be exploited to initiate side channel attacks by inferring UI (user interface) states on mobile devices such as smartphones. We investigate the feasibility of using unprivileged power traces to infer sensitive UI states of mobile applications (*apps* for short), such that the attacker would learn the exact timing to initiate the corresponding attacks.

For example, in order to initiate activity hijacking attacks on Android, attackers need to know when the user login UI will be prompted so that they can intercept the UI state transition and insert fake user login UIs that could steal user credentials. In our work, *we regard the UI states of a mobile app as the confidential information that the attacker wants to gain through side channel attacks; while the power traces, as an unprivileged resource, can be used as a side channel to achieve this goal.*

Our Proposal This paper proposes PoWatt (***PoW***er ***Att***ack), a method to infer sensitive UI states based on power traces collected on Android smartphones, in order to demonstrate the feasibility of power side channel exploitation. Specifically, we investigate the effectiveness of capturing sensitive UI loading events from power traces collected during app execution.

The key idea of PoWatt is based on the fact that *power patterns of each UI loading even in Android apps has unique features that distinguish it from other UI loading events.* We can study the power patterns of a sensitive UI in advance and detect its occurrences on another phone based on the learned pattern. PoWatt is thus designed as a typical pattern matching approach, which involves *training data collection, model training* and *target UI detection.*

During training data collection, we design a method to identify the starting point of a UI loading event and use automated scripts to collect the power trace of a target UI. The power traces are collected with a software-based approach that can read power values from the smartphone profile.

In the model training phase, we generate a prediction model by splitting the training data into different groups and finding the most accurate parameters to generate the most matches between these groups. The result of model training includes fitting curves and accompanying parameters for each sensitive UI .

The detection phase can be conducted either offline or online. In offline detection, we use the trained model to detect the target UI from continuous power traces collected separately from training data collection. The algorithm detects matching target UIs with a time window sliding along the time-indexed power trace. In online detection, the algorithm is the same, while we add the detecting algorithm running in background in the training phase to reduce its impact on the power patterns. We carry out real-time experiments by inviting several volunteers to use the above-mentioned apps with our exploitation tool running in the background.

Results Overview We perform experiments on four popular Android apps, include Alipay, Amazon, WeChat, and Word. Most experiments are conducted on a Nexus 5 smartphone. We collect power traces with automated scripts manipulating these apps traversing different UIs including the target sensitive UIs, and detect the occurrences of these sensitive UIs with PoWatt using the trained models for each app.

In experiments with offline detection, we split the collected data into five groups and performed five-cross evaluation. Results show that we can achieve an average precision of about 71% (up to 98%) and an average recall rate of about 70% (up to 88%) to detect given sensitive UIs For online detection, PoWatt detects 45-85% of the target UI occurrences in real-time cases, with an average precision of 66%.

The results demonstrate that we are able to infer a target UI state from the power trace of a running app with a reasonable precision and recall rate, thus it is practical for attackers to exploit power traces to infer UI states.

Although our approach is not perfect in terms of detection accuracy, it presents a real threat to user privacy as an attacker is able to detect the presence of a particular UI with a reasonable successful rate, revealing that attacking based on power side channel is becoming a practical concern.

2 Background and Motivation

2.1 Power Measurements

The power consumption of a smartphone can be measured with both hardware-based and software-based methods. Although power measurements based on hardware meters are very accurate, it is not applicable to real-world scenarios, thus we use a software-based measurement method to record power traces on a smartphone.

Power related readings are publicly accessible on most smartphone OSes such as Android. In general, instant power numbers can be calculated based on voltage and current readings of BMU (Battery Monitoring Unit)[20]. The battery status information is accessible by most apps without system-level privilege, as many mobile apps need to know the battery status to carry out responses such as saving user context before the battery dies.

The power numbers can be calculated by polling battery status files. Battery device drivers are required to updating these files in order to provide instant power numbers of Android system. Within these files, power consumption is specified in microamps (μA) of current and in microvolts (μV) of voltage. The update frequency varies with different devices, ranging from 10 to 100 times per second.

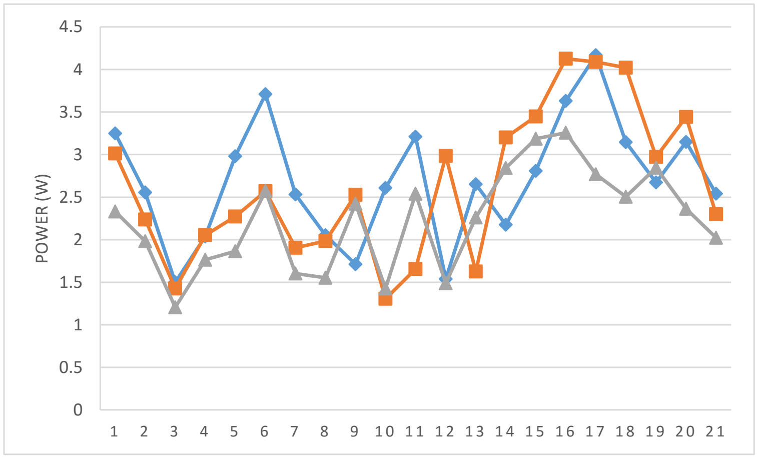

Fig. 1: Power traces collected for the log-in UI for the Amazon app through software-based measurement, in three different runs.

2.2 Distinguishability in Power Patterns

Our study is based on the following hypothesis: *different UIs within an app are distinguishable based on their power patterns* since different UIs have different usage of network communication, calculation tasks and UI rendering.

We use software-based method to measure the user login UI from the Amazon app running on a Nexus 5 smartphone with Android 6.0. Figure 1 shows the power traces of the user login UI in three different test runs. We can see that the power patterns exhibit obvious similarity for the same UI on different test runs, which makes it distinguishable in a continuous power trace.

Based on the measurement results, we observed the existence of power side channels that can be used to distinguish between different UI States, which is potentially exploitable for attackers to infer sensitive UI loading phases such as user login (password input) UIs. This motivates us to conduct further studies on the feasibility of exploiting power side channels to infer sensitive UI states. More details on the measurement study can be found in our earlier work [19].

3 PoWatt Overview

The goal of our study is to demonstrate the feasibility of inferring sensitive UI states of mobile apps through power side channels on mobile devices. In order to achieve this goal, we face the following challenges:

– ***How do we capture the power pattern for a target UI?*** A user may visit dozens (or even hundreds) of different UIs when using a mobile app, thus we need to find a way to specify the particular UI that might be interested to

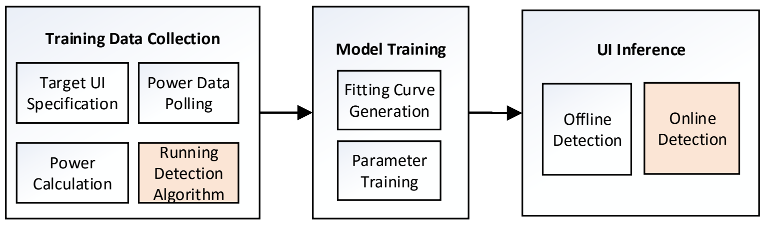

Fig. 2: Overview of PoWatt. (Note that the shaded components are used for real-time online detection only.)

the attacker. We also need to specify the starting and ending points of a UI loading phase before we learn its pattern.

– ***How do we detect the occurrence of a target UI based on its power pattern?*** Even we have obtained a unique power pattern for a target UI, we need to find a way to detect the power pattern in a continuous power trace, as the user will use the app in a continuous manner. Furthermore, the exact power readings and loading time might vary in different occurrences of the same UI, even it runs on the same smartphone.
– ***How do we conduct real-time online detection of the target UI?*** Detecting a target power pattern in an offline power trace may be easier to do, however, performing meaningful attacks typically requires real-time online detection. Online detection increases the complexity because the detection mechanism itself also costs considerable power, which may pollute the power patterns and cause detection to fail.

3.1 Threat Model

In our study, the attacker (i.e., the malicious app) is installed in the same OS environment with the victim app, which contains some sensitive UIs that might reveal its secrets, such as login passwords or financial data. The goal of the attack is to learn the timing of these sensitive UIs when it appears on the screen such that the attacker can perform further actions to steal the secrets.

We assume that the attacker (i.e., the malicious app) has prior knowledge on the victim app and the target UI it attempts to detect. For example, the attacker can install the victim app on another smartphone (preferably of the same model) and collect the power traces to study the power patterns of the target UI state, such as a login screen.

When the malicious app is conducting real-time power side channel exploitation, it runs in the background recording the power data while the victim app runs in the foreground. The malicious app attempts to infer sensitive UI states of the victim app based on the power trace it collects, initiating further attacks once it detects the correct timing.

3.2 Overview of PoWatt

The purpose of PoWatt is to find an effective method to capture the occurrences of the target UI from a continuous power trace collected from an app. Figure 2 presents an overview of PoWatt, which involves the following main steps.

Training Data Collection The first step is collecting training data. We first specify a sensitive UI (i.e. the target UI) within the target app, for instance, the user login UI. Then we run the target app in the foreground and the data collecting program in the background to collect multiple power traces continuously.

In our experiments, we use an automated script to visit different UIs in this app, with the target UI (e.g., the login UI) visited multiple times during the process. The result is a continuous power trace that includes the power patterns of the target UI and other UIs as well.

During *online detection*, running a real-time pattern matching algorithm in the background will increase power consumption and affect the pattern matching accuracy. In order to simulate the same environment as in the detection phase, we also let the same pattern matching algorithm running in the background while collecting power data for the training dataset on the target UI for online detection.

Model Training Based on a subset of the collected power traces (the rest will be used in testing), we then train a model to identify the target UI. These power traces are basically time-indexed power numbers recording the occurrence of the target UI. In order to obtain the fitting curve, we first apply several pre-processing steps including calculating an average curve and smooth the data with moving average to eliminate noises.

After pre-processing, we split the training data into two separate sets. We first calculate a fitting curve with one part of the training data and use it to detect the target UI in the rest of the data. We use a simple genetic algorithm to find the parameters that yield the best overall accuracy (considering both precision and recall). These fitting curves and accompanying parameters will be used to conduct the detection in the next phase.

UI Inference We perform both offline and online detection to demonstrate the possibility of detect the target UI while exploiting the power side channel.

For *offline detection*, we use the model trained with training data to detect the target UI in the testing data. We apply a time-window based pattern matching algorithm with the trained model. Offline detection is used to demonstrate the feasibility of power trace exploitation, thus we do not run the detection algorithm in real-time to prevent it from polluting the power trace.

We also perform *online detection* in a more realistic environment, where the detection algorithm runs in the background on a smartphone to detect the target UI, which runs in the foreground. Because the detection algorithm also consumes power, we run the power trace collection and training process again, with the detection algorithm running in the background. Thus we will have similar power patterns during training and detection. The detection algorithm is the same as used in offline detection.

4 PoWatt Design

4.1 Data Collection

Target UI Specification Mobile apps are composed of different UI components (*i.e., Activities* in Android). In a single app, the user typically navigates through multiple UIs to use some specific app functionality. In Android, current and past UIs are saved and maintained in a stack data structure called a *Back Stack*. When a new UI is loaded, it is pushed on the top of the stack. If the current UI has a "parent" UI (e.g., the UI has a "back" button), it gets popped out the stack when the user returns to its "parent" UI. As a result, when the loading process of some specific UI happens, there are two possibilities of the trigger source: (1) the user creates a new instance of the UI; (2) the user navigates back from one of its "children" UIs.

A typical UI loading process in the Android framework works as follows: (1) The `ActivityManager` component calls `performLaunchActivity()` API; (2) the `onCreate()` or `onPause()` function (both of them are implemented by the app) gets called depending on the source of trigger; (3) After that, the `performTraversal()` API is called, in which the loaded UI will be put into the framebuffer and the screen gets repainted; (4) Finally, if the UI is newly created, it will be pushed into the *Back Stack*, and the current one will be destroyed; if the UI is the parent of the current one and it gets resumed, the current one will be popped out and destroyed.

Each UI loading process is unique because it involves loading different resources in different sequences, and showing different color schemes on the screen.

We modify the Android framework to record power traces of the target UI. When we record the power trace of each UI loading, we use the `performLaunchActivity()` API calling point as the starting point a UI process, and monitor it until the completion of the `performTraversal()` API.

Power Data Polling and Calculation We collect the power numbers of the target UI to form a power trace, which is used to reflect its power characteristics. To collect the power numbers, we use the built-in software-based measuring method. Instant current and voltage numbers can be acquired by polling system battery status files.

Adaptation for Online Detection The above procedure works well for offline detection of the target UI in a continuous power trace. However, when we try to conduct online detection, the detection algorithm itself consumes significant power, which affects the power consumption patterns of the target app.

In order to minimize the influences of the online detection algorithm, we run the algorithm in the background when we collect the training power data. In this way, we simulate the same environment as in the detection phase while collecting power data for the training dataset on the target UI.

4.2 Model Training

During model training, our goal is to generate a representing power pattern (i.e., fitting curves) and accompanying parameters (thresholds), which is used to identify the target UI in a power trace during the detection phase. In our study, we generate the fitting curve based on the power consumption time series $TS_1, TS_2, ..., TS_n$ extracted from the power traces, which contains n runs of the loading phase of the target UI .

After collecting the training dataset, we first calculate an average power pattern based on the set of different power patterns $(TS_1, TS_2, ..., TS_n)$ for the same UI, and then apply a Gaussian filter to smooth the power curve by calculating their moving average. The result is the main fitting curve (FC).

As a supplement to the power traces, we also calculate a *power differential series* (DS) that considers the difference of the power numbers in each adjacent pairs in the trace. For each $TS_i = p_1, p_2, ..., p_n$, its corresponding DS_i is calculated as $p_2 - p_1, p_3 - p_2, ..., p_n - p_{n-1}$. We calculate the average DS and apply the same Gaussian filter to smooth the curve. The resulting differential curve (FC') is used to represent the power trend for the target UI, which we consider as an important supplement to the main fitting curve.

For the generated fitting curve FC, we then calculate the distance of each power pattern to the fitting curve, and use the average distance as the threshold (Th). For the differential curve FC', we calculate a threshold (Th') using the same method.

We can use either fitting curve and the corresponding threshold to detect the target UI in a continuous power trace. However, we want to train a model that involves both fitting curves to achieve better accuracy. For each time series TS_i, we add two parameters P_1 and P_2, and use the following criteria to determine whether is a match:

$$Sigmoid(\frac{Dist(TS_i, FC)}{Th} + P_1 \times \frac{Dist(DS_i, FC')}{Th'}) > P_2$$

where P_1 and P_2 will be trained using a simple genetic algorithm to maximize the F-Measure value:

$$F\_Measure = 2 \times \frac{precision \times recall}{precision + recall}$$

When calculating distances between two time series, we use the square of the actual distance, since the Euclidean norm is better than Manhattan norm in terms of preventing overfitting.

We use a sliding window approach as shown in Figure 3 to detect whether there is positive match in a power trace based on the above criteria. Once we detect the match, the sliding window will jump to the end of the match and continue. The matching algorithm is the same as used later in the UI inference step. The parameters P_1 and P_2 will be initially set as a number from (0.2, 0.5) and (0.5, 0.7), respectively. For the genetic algorithm, we generate 200 instances for each generation and train them over 10 generations, in order to find the best possible parameters (P_1 and P_2).

Please note that the model training process for online detection is the same as for offline detection. The only difference is that the power traces used when training for online detection include the power consumption of the detection algorithms running in the background.

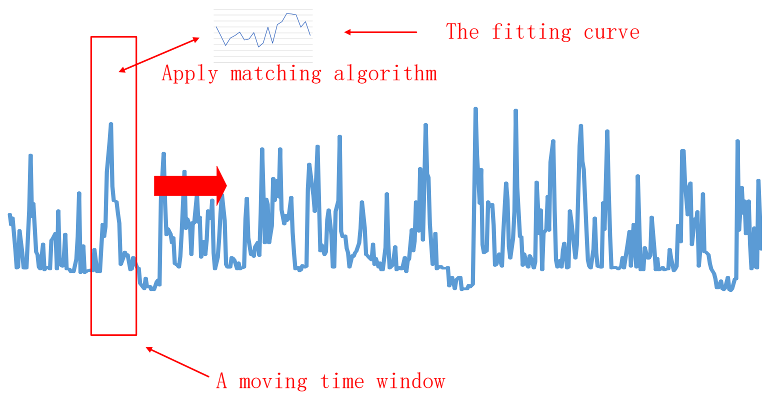

Fig. 3: The workflow of the UI detection algorithm.

4.3 UI Inference

In the UI Inference phase, PoWatt applies the same detection algorithm as used in training based on the fitting curves (FC and FC'), the thresholds (Th and Th'), and the trained parameters (P_1 and P_2), to detect whether there is a match to the target UI in the testing power trace.

Offline Detection For offline detection, the goal is to identify the occurrence of the target UI in a continuous power trace in an offline manner, after we record the power trace.

The detection process is depicted in Figure 3. We create a time window along the time series with window size equaling the longest power pattern for the target UI during traning. Then we apply the detection algorithm mentioned earlier to repeatedly calculate the distance between the fitting curves and the times series in the current window. We find a match when the distances satisfy the given criteria for the trained parameters and thresholds.

Without real-time background noises, offline detection can demonstrate the capability of PoWatt to detect the target UI in ideal situations.

Online Detection The method we adopt in online detection is the same as in offline detection. However, because we apply pattern matching in real-time, it adds extra power consumption to the power trace, such that we need to re-train the model with the power traces collected with the detection algorithm running in the background. Fortunately, the detection algorithm itself is not power hungry and its power consumption patterns is regular. Thus we are still able to find a power pattern to match the target UI even with detection in the background.

Online detection will be performed by volunteers, such that we can demonstrate the capability of PoWatt to detect sensitive UIs in real scenarios.

Table 1: Details of the mobile apps used in our experiments.

App	Version	Category	Target UI
Alipay	9.5.3	Payment	password input
Amazon	6.4.0.100	Shopping	Log in
WeChat	6.3.15	Communication	password input
Word	1.0.1	Productivity	Log in

Table 2: Power trace specification in each automated script. (We ran the same script five times during data collection.)

App	# of Unique UIs	Target UIs	Other UIs
Alipay	9	20	100
Amazon	11	20	100
WeChat	10	20	100
Word	9	20	100

5 Experiments and Results

5.1 Experimental Setup

In our experiments, we choose four popular Android apps from including Alipay, Amazon, WeChat and Word. Details of these apps are shown in Table 1. We consider the user login or payment password input UI of each app as the target UI, as attackers may try to steal user passwords from these apps. We use a Nexus 5 smartphone with Android 6.0 for most of our experiments.

For each app, we write an automated UI testing script based on a technique for building test cases for Android apps [10]. We do not use the popular `MonkeyRunner` here since it requires `adb` connection, which will result in big influence on the power patterns. Within each automated UI testing script, we try to reach multiple UIs while achieving the desired number of occurrences (20 in each trace) of the target UIs.

Table 2 shows the power trace statistics. For each app, we include 20 occurrences of the target UI, as well as 100 occurrences of other UIs (including different unique UIs as list in the table). We use the automated scripts to collect five traces for each app.

For offline detection, we use these automated UI testing scripts to generate 5 power traces for each app, then we conduct five-cross evaluation, each time using four power traces as training data and the remaining power trace as the testing data. We train a model with the four training power traces and use the model to predict target UIs in the testing trace. Offline detection is performed on a desktop PC with the power traces.

In order to evaluate the effectiveness of real-time online detection, we invite volunteers from our lab to use the apps listed in Table 1 on the Nexus 5 smartphone.

With the online detection program running in the background, all the participants are trained to perform two different tests: one using a given UI sequence

Table 3: UI Inference results for offline detection on collected power traces. We show five-cross examination results, and their average.

App	Run #1		Run #2		Run #3		Run #4		Run #5		**Overall**	
	prec.	recall	prec.	recall	prec.	recall	prec.	recall	prec.	recall	**prec.**	**recall**
Alipay	64%	80%	68%	85%	65%	85%	67%	80%	55%	85%	**64%**	**83%**
Amazon	43%	50%	55%	55%	68%	65%	79%	55%	57%	20%	**60%**	**49%**
WeChat	56%	70%	67%	50%	67%	60%	58%	90%	62%	40%	**62%**	**62%**
Word	91%	100%	100%	90%	100%	100%	100%	55%	100%	95%	**98%**	**88%**

which is the same as the in the automated test script; the other asking the volunteer to visit different UIs inside each app with randomized order and number of visits. In both tests, they are asked to visit the target UIs (login UIs) for exactly 20 times. Each participant has 10 minutes to finish the experiments. If a positive matching of the target UI is detected, a notification will be pushed on top of the screen in the notification area to remind the user. We ask all participants to count the number of total positive detection notifications, and whether it is a true positives (or false positive).

5.2 Results and Analysis

Offline Detection Table 3 shows the results of UI inference during offline detection. We perform five-cross evaluation with the five power traces collected for each app. The results include the precision and recall numbers in each test, as well as their average value.

We can see that the Word app has the best overall result in both precision and recall of 98% and 88%, respectively. The reason is because the long in UI of the Word app is implemented as a webpage in WebView, thus its loading time is relatively long. A long loading time will expose more features of the target UI, thus increasing the detection accuracy. For the other three apps, their detection precision are all at about 60%, which is acceptable. In all four apps, Amazon fares the worst with an average recall of only 49%.

Although the results still have space for improvement, they are good enough to be used in meaningful attacks as attacks do not have to be successful every time. We have accomplished our goal to demonstrate the effectiveness of inferring UI states exploiting a power side channel.

Online Detection Table 4 shows online detection results following the same UI sequences as in the automated scripts. Because there are no new UIs introduced during the test, the detection accuracy remains comparable to what we have as for offline detection. On average, we are able to detect the target UI in real-time with an average precision of around 66% and an average recall rate of 59%. The highest detecting precision and recall is on the Word app with 94% and 85%, respectively.

We then show online detection results with random UI sequences in Table 5. The average precision now drops to 43% while the average recall drops to 54%.

Although both the precision and recall rates are lower than those from the previous power trace study, this is expected because real-time pattern matching

Table 4: Results of online detection (on a Nexus 5 smartphone with Android 6.0). The volunteers followed the UI sequence in the automated script.

App	Unique UIs	Target	P	FP	Prec.	Recall
Alipay	9	20	16	6	63%	50%
Amazon	11	20	18	7	61%	55%
WeChat	10	20	20	11	45%	45%
Word	9	20	18	1	94%	85%

Table 5: Results of online detection (on a Nexus 5 smartphone with Android 6.0). The volunteers were free to click as many different UIs as possible.

App	Unique UIs	Target	P	FP	Prec.	Recall
Alipay	45	20	29	20	31%	45%
Amazon	25	20	25	18	28%	35%
WeChat	23	20	26	17	35%	45%
Word	12	20	23	5	78%	90%

brings instability to the power patterns. However, even with the Amazon app, we are still able to detect the timing of user login with a one in three chance. Even there is a 72% chance that we might mispredict, we are still able to perform meaningful attacks with a reasonable success rate.

6 Case Study

To show that our methodology in PoWatt can pose real-world threats on s-martphones, we present a case study of a real-world attack exploiting power side channels. The attack we demonstrate here is a screenshot-based UI attack, which is introduced in ScreenMilker [11]. The attackers could steal user passwords through this attack on smartphones.

We assume that the attack happens in an environment of common configurations where the Android OS does not have be compromised and the malicious app is a totally legit non-system app with no extra permissions needed (we may need the "network" permission to broadcast the attacking results, but it is in fact unnecessary). The malicious app and the victim app are co-installed on the same Android OS, and our case study show that the malicious app could successfully steal confidential information from the victim app with the hints provided by PoWatt.

We assume that the attacker has prior knowledge of the victim app and has already generated a detection engine using the online detection model training techniques described in PoWatt. The malicious app then runs in the background and collects the power trace continuously in real-time. While collecting the power data, it continuously applies real-time detection to check whether the user has attempted to load the target UI (login UI in this example). Once it detects that the login UI has been loaded, the malicious app starts to take screenshots of the

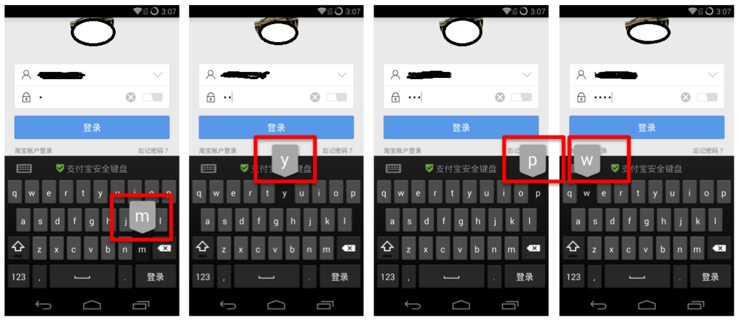

Fig. 4: The attacker steals user credentials by continuously taking screenshots of the Alipay login UI after detecting the timing of its loading.

victim app in order to steal sensitive information (i.e., username or passwords) from it.

We implemented our attack on a Nexus 5 smartphone and choose the Alipay app as our victim app. Alipay is a popular mobile payment app with multiple functions including payment, money transfer and investment. We consider the user passwords of Alipay app to be highly sensitive.

Figure 4 shows the screenshots taken in our case study. Once the malicious app detects the loading of the user login UI, it will continuously take screenshots of the victim app. With the default prompt of the keyboard animation on tapped keys, the attacker is able to steal the user's Alipay password ("**mypw**"). Attacker could achieve this either programmatically with some graphical recognition algorithms or manually with full images acquired.

Although the attack is pretty straightforward to apply, the most important thing in the attack procedure is that the attacker has to know when to start taking screenshots. Although the login UI for the Alipay app is not guaranteed to be detected each time it loads, with several more attempts, the attacker will eventually get the chance to detect the occurrence of the target UI and capture the desired passwords successfully.

Please note that in this attacking example, the malicious app requires extra permissions to take screenshots, which is not considered a very sensitive permission in Android as many apps are allowed to perform the action. However, when the seemingly innocuous privilege is exploited together with power side channels, the attacker can successfully steal sensitive information from the apps.

7 Discussions

Threats to Validity We have demonstrated that we are able to infer sensitive UI states with power side channels and perform real-time attacks to steal user information. There are a few limitation of this work that might affect its validity.

Power side channels might only be distinguished for a limited set of UI operations whose power patterns are consistent each time it is loaded. Some other UIs might exhibit different power patterns each time it is loaded. For example, an image display app may consume different power while loading image thumbnails if the number of images it processes are different. Fortunately, we observe that sensitive UIs involving login passwords or financial data are typically stable and exhibit unique power patterns.

Possible Mitigations In order to protect users from power side channel attacks, we could make modifications to mobile apps or the OS itself.

– **Energy obfuscation through code injection.** One straightforward mitigation approach is that we can inject meaningless code into mobile apps while performing sensitive user interactions, in order to insert power bursts into its power pattern to make it unpredictable. This can be achieved at the source code level during the app development process, or through instrumentation to the bytecode for app binaries.
– **Randomly changing display/color parameters.** One interesting feature for the OLED or AMOLED displays used for smartphones is that it consumes different power when different color schemes are used [5]. Thus we can vary the displaying color and other parameters each time the sensitive UI is displayed on the screen. This could be achieved during app development or through bytecode instrumentation [9].
– **Raising the privilege needed to access power files.** Of course, we can always make the power information privileged, such that not all apps could access these data directly. As a matter of fact, mobile apps probably do not need to read low-level power related files containing raw voltage or current readings. The only thing that most apps need to know is how much battery is still remaining, which should not pose serious threats as a side channel.

8 Related Work

8.1 Power Side Channels

Power analysis attacks (or power side channels) [1] have become an important type of side channel attacks in recent years. One well-known example of power analysis is the recovery of an encryption key from a cryptosystem [8, 7]. Messerges *et al.* [14, 13] examined both simple power analysis(SPA) and differential power analysis (DPA) attacks against the data encryption standard (DES) algorithm and managed to breach the security of smart-cards using the proposed signal-to-noise ratio (SNR) based multi-bit attack.[12]

On mobile platforms, Michalevsky *et al.* proposed PowerSpy [15], which investigates the relation between signal strength and the power pattern of the smartphone and showed that they can infer smartphone users' whereabouts based on the power traces.

Our work also focuses on the mobile platform, but we have presented a different and more general attack in UI state inference based on power traces.

8.2 UI-based Attacks

The UI security of an application has been studied extensively [17, 3, 6]. On traditional desktop platforms, UI-based attacks are basically categorized as UI spoofing attacks [3, 6]. Recently, UI-based attacks start to emerge on mobile platforms. For example, ScreenMilker [11] can take screenshots of the foreground app covertly and steal user credentials.

Chen *et al.* propose an attack on the Android platform called UI inference attack [2]. They use the share-memory side channel to infer UI states, in order to detect the correct timing for attacks. Our work targets at a similar attack in UI inference, but we have achieved it through power side channel exploitation.

9 Conclusions

In this paper, we present PoWatt, a method that demonstrates the existence of a new side channel to infer UI states of mobile apps: the *power side channel*. Attackers can infer the UI states of a mobile app in the foreground with an un-privileged app running in the background, which helps to identify the timing of attacking on sensitive user inputs or screen outputs based on power traces.

The results demonstrate that we are able to infer a target UI state from the power trace of a running app with a reasonable precision and recall rate, thus it is practical for attackers to exploit power traces to infer UI states. Although this study on power side channels is only a small step towards understanding the power side channel issues on mobile devices, it shows that there are new ways to perform attacks based on unprotected power information. More studies are needed to investigate its potential damages and possible mitigation techniques.

Acknowledgments

This work was partly supported by the National Natural Science Foundation of China (No. 61772042).

References

1. E. Brier, C. Clavier, and F. Olivier. Correlation power analysis with a leakage model. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 16–29. Springer, 2004.
2. Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: UI state inference and novel android attacks. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, pages 1037–1052, 2014.
3. S. Chen, J. Meseguer, R. Sasse, H. J. Wang, and Y.-M. Wang. A systematic approach to uncover security flaws in GUI logic. In *IEEE Symposium on Security and Privacy, 2007. S&P '07*, pages 71–85. IEEE, 2007.
4. S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 191–206, 2010.

5. M. Dong and L. Zhong. Power modeling and optimization for OLED displays. *IEEE Transactions on Mobile Computing*, 11(9):1587–1599, Sept 2012.
6. T. Fischer, A. Sadeghi, and M. Winandy. A pattern for secure graphical user interface systems. In *The 20th International Workshop on Database and Expert Systems Application, 2009. DEXA '09*, pages 186–190, Aug 2009.
7. P. Kocher, J. Jaffe, and B. Jun. Introduction to differential power analysis and related attacks. *http://www.cryptography.com/resources/whitepapers/DPATechInfo.pdf*, 1998.
8. P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology (CRYPTO '99)*, pages 388–397. Springer, 1999.
9. D. Li, A. H. Tran, and W. G. J. Halfond. Making web applications more energy efficient for OLED smartphones. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 527–538. ACM, 2014.
10. Y. Li, Z. Yang, Y. Guo, and X. Chen. Droidbot: A lightweight ui-guided test input generator for android. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 23–26, 2017.
11. C.-C. Lin, H. Li, X. Zhou, and X. Wang. Screenmilker: How to milk your android screen for secrets. In *Proceedings of The 21th Annual Network and Distributed System Security Symposium (NDSS)*, 2014.
12. S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.
13. T. S. Messerges, E. A. Dabbish, and R. H. Sloan. Power analysis attacks of modular exponentiation in smartcards. In *Cryptographic Hardware and Embedded Systems*, pages 144–157. Springer, 1999.
14. T. S. Messerges, E. A. Dabbish, and R. H. Sloan. Examining smart-card security under the threat of power analysis attacks. *IEEE Transactions on Computers*, 51(5):541–552, 2002.
15. Y. Michalevsky, G. Nakibly, A. Schulman, and D. Boneh. PowerSpy: Location tracking using mobile device power analysis. In *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C., Aug. 2015.
16. Z. Qian, Z. M. Mao, and Y. Xie. Collaborative TCP sequence number inference attack: How to crack sequence number under a second. In *ACM Conference on Computer and Communications Security*, CCS '12, pages 593–604, 2012.
17. J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the EROS trusted window system. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, pages 12–12, 2004.
18. J. C. Wray. An analysis of covert timing channels. In *Proceedings of 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 2–7, 1991.
19. L. Yan, Y. Guo, X. Chen, and H. Mei. A study on power side channels on mobile devices. In *Proceedings of the Seventh Asia-Pacific Symposium on Internetware (Internetware'15)*, 2015.
20. C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. AppScope: Application energy metering framework for android smartphone using kernel activity monitoring. In *USENIX Annual Technical Conference*, pages 387–400, 2012.
21. D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 563–574. ACM, 2011.
22. L. Zhuang, F. Zhou, and J. D. Tygar. Keyboard acoustic emanations revisited. *ACM Transaction Information System Security*, 13(1):3:1–3:26, Nov. 2009.