OS-level Side Channels without Procfs: Exploring Cross-App Information Leakage on iOS

Xiaokuan Zhang*, Xueqiang Wang†, Xiaolong Bai‡, Yinqian Zhang*§ and XiaoFeng Wang†
*{zhang.5840,zhang.834}@osu.edu, The Ohio State University
†{xw7,xw48}@indiana.edu, Indiana University at Bloomington
‡bxl12@mails.tsinghua.edu.cn, Tsinghua University

***Abstract*—It has been demonstrated in numerous previous studies that Android and its underlying Linux operating systems do not properly isolate mobile apps to prevent cross-app side-channel attacks. Cross-app information leakage enables malicious Android apps to infer sensitive user data (*e.g.*, passwords), or private user information (*e.g.*, identity or location) without requiring specific permissions. Nevertheless, no prior work has ever studied these side-channel attacks on iOS-based mobile devices. One reason is that iOS does not implement `procfs`—the most popular side-channel attack vector; hence the previously known attacks are not feasible.**

In this paper, we present the first study of OS-level side-channel attacks on iOS. Specifically, we identified several new side-channel attack vectors (*i.e.*, iOS APIs that enable cross-app information leakage); developed machine learning frameworks (*i.e.*, classification and pattern matching) that combine multiple attack vectors to improve the accuracy of the inference attacks; demonstrated three categories of attacks that exploit these vectors and frameworks to exfiltrate sensitive user information. We have reported our findings to Apple and proposed mitigations to the attacks. Apple has incorporated some of our suggested countermeasures into iOS 11 and MacOS High Sierra 10.13 and later versions.

I. Introduction

Android and iOS are the two most popular operating systems (OS) used in smartphones, wearables, and tablets. Security of such mobile systems has been widely studied in the past decade, mostly in the context of Android [31], [36], [42], [45], [66], [67], with some limited effort on iOS [34], [37]. Of particular interest here are a series of *side-channel attacks*, which empower an untrusted third-party app (*e.g.*, free games) to *infer* private user information by monitoring the execution of OS services or trusted apps (*e.g.*, banking apps). Side-channel attacks typically do not exploit software vulnerabilities to acquire secret data directly. Instead, confidential user information is inferred from *vectors* (features of the target as observed by the adversary) that are considered harmless but actually reveal some artifacts of the target app or service's executions. Examples of such vectors include the use of shared CPU caches (*i.e.*, *cache side channels*), mobile sensors (*i.e.*, *sensor-based side channels*), and public APIs provided by the OS to third-party apps for querying the status of the mobile device, the OS, or other apps (*i.e.*, *OS-level side channels*).

This paper focuses on OS-level side channels. So far, this line of research has only been conducted on Android, with numerous studies [31], [36], [42], [45], [64], [67] showing that the OS and its underlying Linux kernel fail to properly control the information leaks from *seemingly harmless* sources—`procfs`, a pseudo filesystem available on UNIX-like operating systems (including Android) to export some kernel statistics (*e.g.*, virtual and physical memory, CPU and network usage) to the user space. These statistics can be classified into two categories: per-process statistics and global statistics. Per-process statistics reveals the information pertaining to a specific process, while global statistics reports aggregated information from all processes and the entire kernel. Most existing side-channel attacks exploit per-process statistics in `procfs` [31], [42], [45], [64], [67].

Unlike Android, iOS is known for its more aggressive security controls, which render many attacks that succeed on Android less likely to happen on iOS. Specific for the OS-level side-channel threats, the iOS kernel is built on top of Mach [22] and FreeBSD [38], which does not have a `procfs`, essentially shutting down the main avenue for the Android-style inference attacks. Although a small amount of `procfs`-alike resources are still available on iOS (*e.g.*, through `sysctl()`) [15], they are under heavy scrutiny and facing increasingly stringent restrictions: as a prominent example, since iOS 9, Apple has modified `sysctl()` to disallow a sandboxed app from accessing information about other running processes [16]. As a result, it is impossible today to conduct a side-channel analysis on iOS by exploiting per-app statistics, which completely defeats most demonstrated attacks on Android [31], [42], [45], [64], [67]. Therefore, to our knowledge, there was no reported OS-level side-channel attack on iOS.

As such, in this paper, we make the *first* step towards understanding cross-app side-channel risks on iOS. More specifically, we started with an inspection of new attack vectors on iOS. Our study has led to the discovery of several APIs reporting global statistics of the entire system, including CPU usage, memory usage, network usage, and storage usage, with examples of a subset of them described in Sec. III. These

§Corresponding author.

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23260
www.ndss-symposium.org

findings, though new to the research community, are not extremely surprising, as such functionalities are supposed to be provided by the OS to the userspace. Also, as one can anticipate, these global statistic counters are noisier than per-app statistic counters; when used individually in side-channel attacks, they do not expose sufficiently intelligible information about a specific target (e.g., a process). However, our study shows that collectively, these counters can actually be utilized together to deduce surprisingly detailed user data, once we have addressed the machine learning challenges in integrating the information from these individual sources, which have never been considered in previous side-channel studies.

Specifically, we developed machine learning frameworks that combine multiple *noisy* side-channel attack vectors in a *novel* way. Particularly, we designed a classification framework that samples time series of the data from 6 global statistic counters, reduces their dimensions and further extracts their key features from training traces by using Symbolic Aggregate approXimation (SAX), Bag-of-Pattern (BOP) representation, and Support Vector Machine (SVM) classifiers. Also we developed a pattern matching framework that employs a kNN classifier with a multi-dimensional Dynamic Time Wrapping (DTW) algorithm to calculate distance metrics. Our evaluation demonstrates that these frameworks are effective (high accuracy), efficient (short execution time), and robust (*e.g.*, models trained on one device can be used on other devices, as shown in Sec. VII).

More concretely, we demonstrate three categories of attacks on iOS 10, the latest iOS version as of the time of writing this paper: classifying user activities, detecting sensitive in-app activities, and bypassing iOS sandbox restrictions to infer cross-container file existences using a timing-based side channel. Specifically, we found that an unprivileged malicious app is able to accurately identify the foreground running apps, the websites *Safari* visits, and the location searched through Apple *Maps*. It can further collect enough information to link Bitcoin addresses, *Venmo* users, and *Twitter* users to a device, and identify a set of installed sensitive apps that reveal private information about the user, *etc*.

Although the focus of our study is iOS, our findings raise a broader question important to the design of the operating systems hosting mutually-distrusting entities: *What is the proper means to isolate these entities and prevent cross-app side channel leaks, given the huge amount, complex interfaces between them?* As we can imagine, this question will be very important to iOS, to Android, and to other platforms such as clouds and IoT frameworks.

Responsible disclosure and Apple's adoption of our countermeasures. In May 2017, we reported our demonstrated side-channel attacks to Apple, which, slightly to our surprise (given the conventional attitudes from other vendors towards side channels), attached high importance to our findings and assembled a team of engineers from different groups to specially work on mitigations of these side-channel threats for the next iOS release. We had several technical meetings with these engineers and discussed several solutions to the problems. As we will detail in Sec. VIII, some of the countermeasures have been adopted in iOS/MacOS to defend against our attacks. We are glad to see that Apple is seriously committed to mitigating these threats by making several major updates in the OS kernel. The threats have been fully addressed in iOS 11.1 and macOS High Sierra 10.13.1.

Contributions. In summary, our paper makes the following technical contributions:

- *New attack vectors.* We identified several iOS APIs that can be exploited for side-channel inferences, which suggests that even on an OS without `procfs`, it is very challenging to eliminate all vectors for cross-app information leaks.
- *New attack methods.* We developed new frameworks to integrate the thin information recovered from individual vectors into serious side-channel leaks, by leveraging a set of machine learning techniques. We also demonstrated the robustness of our approach by training and testing on different devices.
- *New targets.* We presented the attacks on a set of targets never exploited in previous side-channel studies, such as location inference through map loading, user identification via Bitcoin transaction correlation, *etc*.
- *Proposed countermeasures integrated in iOS and MacOS.* Through responsible disclosure and technical discussions with Apple, some of our proposed countermeasures have been integrated into iOS 11.1 and macOS High Sierra 10.13.1.

Roadmap: Sec. II summarizes the background of iOS cross-app isolation. Sec. III highlights our threat model and lists several new side-channel attack vectors on iOS. Three attacks exploiting these attack vectors were presented in Sec. IV, Sec. V, and Sec. VI. Practical issues related to the attacks are discussed and evaluated in Sec. VII. We discuss and evaluate several countermeasures to the demonstrated attacks in Sec. VIII, and summarize related work in Sec. IX. Sec. X concludes the paper.

II. Background: iOS Cross-App Isolation

Side channels on Android mobile devices have been extensively studied in the past [31], [36], [42], [45], [66], [67], however, little or even no attention has been paid to iOS cross-app side channels. In this section, we first briefly introduce the cross-app isolation on iOS. We then describe how side-channel attacks on iOS devices are different from those on Android, and why they are more challenging to conduct in practice.

Sandboxing with respect to file access. Each iOS app is by default confined in a sandbox at installation. A sandbox specifies how an app is allowed to access filesystem resources and communicate with other apps or interact with the operating system. Particularly, the app is only allowed to access files in its own bundle container directory and a few other public directories [3]. The path name of each bundle container directory contains a UUID, which is a 32-digit random hexadecimal string (*e.g.*, `7E698227-C8B6-4044-A215-B4CBCB8A97AB`). Cross-container file accesses are prevented by both the randomness of the UUIDs and the sandbox isolation.

System resources. The `Info.plist` file of an application describes system resources that are needed for an application to run properly. The first time an app attempts to access certain sensitive system resources (*e.g.*, Location Services), the user is

asked to grant the permission explicitly. Only apps authorized by the user are allowed to access the specific resource. This user-centric access control can be configured on a per-app basis. Since iOS 8, Apple has introduced finer-grained access control policies to some system resources. For instance, users can control an app to access Location Service at any time (*i.e.*, *Always*) or only when the app is in the foreground (i.e., *While Using*) [7].

Cross-app communication. Apps commonly communicate with each other through `schemes`. An app may register a custom URL scheme with the system (through its `Info.plist` file). Other apps may use `openURL` API to send data to the app who has registered the custom URL. For example, URL scheme "comgooglemaps://?center=[Latitude,Longitude]" will launch *Google Map* and navigate the maps' center to the specified location. Another means to share data between apps is the pasteboard. The `general` pasteboard provides system-wide read/write access to all apps, while the `named` pasteboard is only accessible by apps with same team ID.

App vetting. Information leakage may happen when iOS APIs are used in unexpected manners or when some undocumented APIs are used by a malicious app. To defeat these API-misuse attacks, all iOS apps, before reaching the market, must be vetted by Apple, who will examine both the functionality of the app and its potential malicious activities [21]. Although Apple's code review process is kept private and continuously changing, it is believed this process includes determining whether private APIs are used by the submitted apps [28], whether private data is collected and transmitted without notifying the users [21], *etc*. Apps that fail the vetting will be rejected by Apple.

III. Side-Channel Attack Vectors on iOS

In this section, we describe the threat model considered in this paper, and new attack vectors that we have discovered to enable OS-level side-channel attacks on iOS without `procfs`.

A. Threat Model

In this paper, we *only* consider side-channel information leakage on the OS level. That is, we aim to explore the API interfaces that allow one iOS app to query information regarding the entire OS or a particular app running on the same device (*e.g.*, iPhones and iPads), and the methods to exploit the leakage to infer private information about the user of the device. More particularly, we assume that the user downloads a monitoring app from iOS App Store. As will be discussed in Sec. VII, our monitoring app disguises itself as an audio player, and registers the `Audio` background mode in its `Info.plist` file to run in the background. No additional permission request needs to be made at runtime. We will show how this monitoring app can utilize some OS-level side-channel attack vectors on iOS (to be discussed shortly) to breach user privacy. Out of the scope are CPU cache side channels [66], electronic magnetic side channels [24], [39], [40], and mobile sensor based side channels [50], [52], [53], [57]. As they explore leakage through micro-architectures, electronic magnetic emission, or device orientation, which are not specific to iOS.

B. New Attack Vectors

We have identified several *new* attack vectors on iOS that enable cross-app information leakage. Particularly, these vectors will allow an iOS app to learn the global usage statistics of memory and network resources, and the existence of files without any access permissions.

Memory resources: host_statistics64(). The global usage of memory resources, such as the number of free memory pages (`free_count`) and the cumulative number of page faults (`faults`), can be queried through this API. Apps do not need special entitlements to access `host_statistics64()`, which is an interface used by iOS apps to access memory information of the current device. We statically analyzed 7,418 iOS apps[1] using a static tool based on Capstone [6], and found that 1,230 of them include this API.

Network resources: getifaddrs(). The usage of the global network resource can be queried through this API without special entitlements. The `getifaddrs()` API returns a linked list data structure describing each of the network interfaces of the local system, by storing the address of the first item of the list in `*ifap`, the argument passed to the API. One can iterate through `ifap->ifap_next` to enumerate all the interfaces. For each item in this linked list, one can read `ifap->ifa_data` to learn the statistics of the traffic that goes in or out through this interface. In particular, we collected the traffic of `en0` (WIFI interface) for analysis in Sec. IV. Other active interfaces include `lo0`, `ipsec0`, `pdp_ip0`, *etc*. `getifaddrs()` is widely used by both app developers and third-party libraries. In our static analysis, we found 3,955 out of 7,418 apps include the API in the apps. iOS apps use this API in many different ways. For example, an app could collect network-related information and upload it to a remote server for crash/error reporting purposes. It may also use the MAC/IP as an identifier of the device.

File Systems: [NSFileManager fileExistsAtPath:]. When an app has proper permission to access a file or directory, the API will return to the caller whether the queried file or directory exists. However, when an app does not possess the required permission, the return value will always be `FALSE`. This sandbox rule protects sensitive files from being accessed by unprivileged third-party apps. [NSFileManager fileExistsAtPath:] is a frequently used Objective-C API, which is referenced by 7,331 of 7,418 apps. For example, to avoid exceptions, an app may check whether a file exists before reading from it.

However, we found that this protecting mechanism can be circumvented using a timing channel. Though the result will always be `FALSE` when the caller doesn't have proper permission, the execution time of this API varies vastly. When the file or directory actually exists, the function call will execute much slower than the cases where the file or directory doesn't exist at all. We conjecture this is because when a file exists, additional permission checks will incur. This execution time difference is big enough to be measured using the API `mach_absolute_time()`. Therefore, one could utilize this timing channel to tell whether a file or directory exists, regardless of the sandbox isolation. In Section VI, we

[1]These apps were sampled from 26 categories in Apple's App Store; they were updated between Jan. 1, 2016 and Feb. 23, 2017.

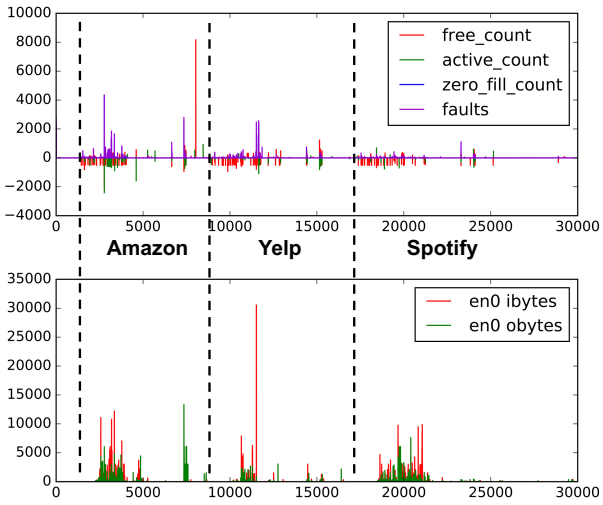


Fig. 1: Traces of global statistics when three iOS apps, *Amazon*, *Yelp*, and *Spotify* were launched and suspended in a sequence.

show that this side channel results in severe privacy leakage on iOS.

IV. Attack 1: Classifying User Activities

In this section, we show how a malicious iOS app may exploit these attack vectors to classify the user's activities on iOS devices.

We exploited 6 features revealing the global statistics of memory and network resources in the attacks. These features are listed in Table I. Specifically, we collected data from 4 features that describe statistics of virtual and physical memory usage (*i.e.*, category `VM`), including `free_count` (the number of free physical memory pages currently available in the operating system), `active_count` (the total number of pages currently in use and pageable), `zero_fill_count` (the number of zero-fill pages), and `faults` (the cumulative number of page faults). These data were collected by repeatedly calling `host_statistics64()`. By extracting data from `getifaddrs()` in similar manners, we collected data from 2 features describing network usage (*i.e.*, category `NW`): `ibytes` and `obytes`, which report the cumulative number of bytes received and sent, respectively, from the Wifi interface *en0*.

Category	Feature
VM	free_count
	active_count
	zero_fill_count
	faults
NW	en0 ibytes
	en0 obytes

TABLE I: Attack vectors.

Fig. 1 shows the data traces of these 6 features (with each point showing the difference between two consecutive data points of the raw data) collected by an iOS app running in the background, when three apps, *Amazon*, *Yelp*, and *Spotify*, were launched and then suspended (pressing the `Home` button) in a sequence. Each data point in the figures shows the value change of the feature compared to the last reading. The APIs were called periodically at the frequency of 1000 times per second. We can see from the figure that the user's activities can be roughly identified by eyeballing the traces of memory and network statistics. More specifically, when an app is launched, the total number of free physical memory pages (*i.e.*, `free_count`) drops, the page fault count increases dramatically in a short period, and intense inbound and outbound network activities are observed (see Fig. 1). When the `Home` button is pressed, typically some memory (*e.g.*, increased value of `free_count`) and network activities can be observed.

These results suggest that it is possible to infer user's activities from these global statistic counters. However, unlike previously demonstrated side-channel attacks that sample data from `procfs` per-app statistics, global counters are much noisier. By eyeballing the traces from data collected from multiple runs of the same experiments, it seems very challenging to exploit any of these features alone to successfully identify the user's activities related to a specific app. Indeed, as we will show in Fig. 3, none of these features can be used individually to achieve high accuracy in the classification of the mobile user's activities. As such, a classifier that combines these features together must be built.

Prior studies that analyze side-channel data traces for inference attacks typically only utilize one data trace to perform the attacks [32], [36], [41], [48], [50], [53]. In these attacks, a single side-channel trace is typically sufficient for the intended attack goals. The challenge we face when conducting side-channel analysis with multiple side-channel traces (with each trace being a time series of data points) is to reduce the dimension of data for classification. Towards this end, we designed a new classification framework to perform side-channel inference attacks with multiple data traces.

A. Attack Methods

We developed a classification framework that maps a set of l time series of n side-channel observations to a "label", L, which corresponds to some user's activity on the device. More formally,

$$\{X_t^1, X_t^2, \cdots, X_t^l\} \Rightarrow L$$

where $X_t^i = (X_{t_1}^i, X_{t_2}^i, \cdots, X_{t_n}^i)$. Our classification framework consists of three major components: SAX, BOP, and SVM, which will be explained in details below.

1) SAX: Symbolic Aggregate approXimation (SAX) was invented by Keogh and Lin in 2002 [54]. It enables us to encode the time series in an efficient way, without losing important information. It requires a time series as input, and it will output a string that represents this time series. The basic workflow of SAX is:

(a) Dimensionality Reduction. To reduce an n-dimensional time series to an m-dimensional time series, the n data points are divided into m equal-sized windows, with each window containing $p = n/m$ data points. The mean value of the data points within a window is calculated and a vector of m such values becomes the new representation of the original time series. This approach is also called Piecewise Aggregate Approximation (PAA) [14].
(b) Z-normalization. The second step is to convert the PAA representation into a series of numbers that follows Normal Gaussian Distribution, i.e., N(0,1). Particularly, let the mean and standard deviation of the time series X_{t_i} be

$\hat{\mu}$ and $\hat{\sigma}$, Z-normalization is performed as $X'_{t_i} = (X_{t_i} - \hat{\mu})/\hat{\sigma}$. [20]

(c) Discretization. Since the time series now follows Normal Gaussian Distribution, it is easy to determine the breakpoints that will produce α equal-sized areas in the Gaussian curve, where α is the number of different symbols used to represent the data, thus a parameter of the framework. For example, Table II shows the corresponding breakpoints when α equals to $3, 4, 5$. Then, according to these breakpoints, the time series of real values is further converted into a time series of symbols. For instance, the values that are smaller than the smallest breakpoint will be replaced by symbol a, the values between the smallest and the second smallest breakpoint will be converted into b, *etc*. Fig. 2 shows an example of SAX when $n = 50$, $p = 5$, $\alpha = 3$. The corresponding SAX string would be $cbbcccbaaa$.

After the process of SAX, we will have a string of m symbols that represents the original time series. The final length of the SAX string can be adjusted by changing the window size p. For example, when $n = 100$ and $p = 4$, the final string length is 25; when $n = 100$ and $p = 10$, the final length is 10.

α	Breakpoints
3	-0.43, 0.43
4	-0.67, 0, 0.67
5	-0.84, -0.25, 0.25, 0.84

TABLE II: Breakpoints.

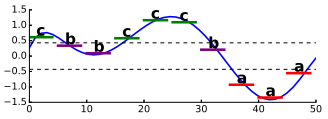


Fig. 2: SAX example.

2) BOP: Bag-of-Patterns (BOP) representation was proposed by Lin and Li in 2009 [46]. BOP converts an SAX-produced string into an array of fixed length. To do so, first a dictionary of all w-symbol SAX strings (dubbed *words*) is created, where w is a parameter of the algorithm. For example, for $\alpha = 2$ and $w = 2$, the size of the dictionary would be $\alpha^w = 4$, and the dictionary of words are aa, ab, ba, bb. Then given the dictionary, BOP counts the frequency of different words in the original SAX string. To avoid over-counting *trivial matches* [46], *i.e.*, similar words that are neighbors in the original SAX strings, we count only the first occurrence of each word. For example, given $\alpha = 2$ and $w = 2$, for an SAX string $aabaabbbbb$, the final result would be: $\{aa : 2, ab : 2, ba : 1, bb : 1\}$. Note that bb is counted only once. The result of this step is called "Bag-of-Patterns" for this time series. For a collection of time series, we can use the *same* dictionary to construct BOP for all of them. Therefore, the converted BOP array is of the same size.

3) SVM: The Support Vector Machine (SVM) is one of the most popular classifiers. In this paper, we choose LibSVM [30] as the tool to perform SVM classification because of its popularity and easy-to-use command-line interface.

Because our attacks will utilize multiple side-channel attack vectors, the input of the SVM classification will have multiple BOP sequences. We concatenate these sequences into one BOP array, which we call the final BOP representation, and then convert it into LibSVM input format. Though the dimension of the final BOP representation may be quite large, only a small fraction of the data points would have non-zero values (less than 10%).

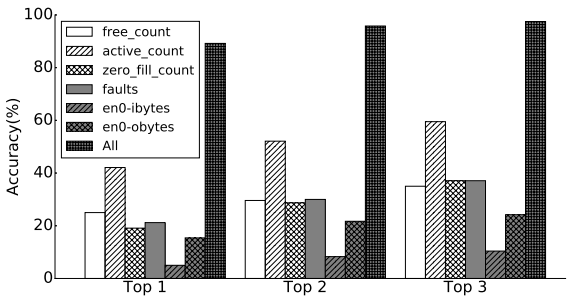


Fig. 3: App classification result using single features.

4) Our Framework: Our framework assembles the three parts mentioned above. The overall workflow is:

(a) *Data Collection*. First, we collect multiple traces for each event that we are interested in. Because the traces are gathered from more than one features, we will have multiple (*i.e.*, l) time series for each trace.

(b) *Difference Calculation*. As we are only interested in the changes of the time series instead of the absolute value, for each time series seq_k ($k = 1, 2, \cdots, l$), we calculate the difference between every two consecutive data points, i.e., $\text{diff}_k[i] = \text{seq}_k[i] - \text{seq}_k[i-1]$.

(c) *SAX Transformation*. By choosing appropriate α and p, we convert the l *diff* sequences into the SAX representation as mentioned in Sec. IV-A1.

(d) *BOP Construction*. After a *diff* sequence is converted into an SAX string, we can construct the BOP of the sequence based on the chosen α and w using the method mentioned in Sec. IV-A2. Because we use the BOP method, there is no need to align different time series.

(e) *SVM Classification*. The last step is to convert the BOPs into LibSVM inputs and use LibSVM to perform classification. We choose the RBF kernel for SVM and use the probability model, which will perform cross-validation and output confidence estimations of classifying a test sample into a class [12].

An alternative approach to our framework is to employ Dynamic Time Wrapping (DTW) [25] to measure distances between samples of multiple classes and use kNN classifiers [33] to perform the classification based on the calculated distances. Our SAX+BOP+SVM framework outperforms this alternative approach because, according to Lin *et al.* [46], when DTW is used as distance measures for large datasets, the time complexity is prohibitive (up to two magnitudes higher time complexity). We have evaluated and compared the execution time of these two algorithms in our own experiments. The results suggest the kNN classifier with DTW is too slow in our setting. We encourage interested readers to refer to Sec. VII-F.

B. Case Studies

We demonstrate how an iOS app can infer the user's activities using the classification framework we laid out in Sec. IV-A in three examples: inferring foreground apps, website fingerprinting, and inferring map searches. These attack targets are of interest to advertisement providers as they can help profile mobile users but are not directly attainable, as iOS disallows third-party apps to learn which app is running in the

foreground, which website the user visits, and which location the user searches, for privacy concerns.

In the attacks that follow, the 6 features in Table I were sampled periodically at the frequency of 1000 times per second. The parameters of the classification framework were selected as $p = 5$, $\alpha = 5$, and $w = 5$. These attacks were conducted on a jailbroken iPhone 7 that runs iOS 10.1.1. Note the attacks do *not* need a jailbroken device to succeed. Using jailbroken device merely made data collection easy to conduct, so that we can apply the classification framework to analyze larger datasets. In Sec. VII-D, we will show that the training and testing of the classifier do not need to be on the same device—the training can be done on a jailbroken device while the attack can be conducted on a non-jailbroken one.

1) Foreground Apps: We downloaded 100 popular apps in April 2017 from the Top charts of free apps in the App Store, and chose another 20 pre-installed iOS apps for the experiment. For each of these 120 apps, 10 side-channel traces were collected, with each trace consisting of 6 time series formed by data sampled from the 6 features in Table I. The sampling frequency was 1000 per second. The monitoring app started data collection before the target app was launched, but only the first 5000 data points after the app launching were included in the time series. It is very easy to programmatically identify the beginning of the app launching procedure as it is quite evident in the traces (as seen in Fig. 1). The target app was terminated after each round of the experiment. We automated the above experiment using *Cycript* [8] on the jailbroken device. We collected in total 1200 traces (*i.e.*, 1200×6 time series of side-channel data) for the 120 apps.

Using the classification framework described in Sec. IV-A, we randomly select 8 traces for each app as the training dataset and the rest 2 traces for each app as the test set. Therefore, there were 960 traces in total in the training set and 240 traces in the test set. Given a test trace, the SVM classifier (using LibSVM) provides a probability estimation of each class it may belong to—the higher the probability, the more likely the trace belongs to the corresponding class. A correct classification means the k^{th} class test sample was correctly classified as the k^{th} class; all other results are considered incorrect. In this way, we can rank the classification results by their probability values and evaluate the top 1, top 2 and top 3 accuracy. Top N accuracy is the percentage of the test samples being correctly labeled by one of the top N predicted classes by the classifier.

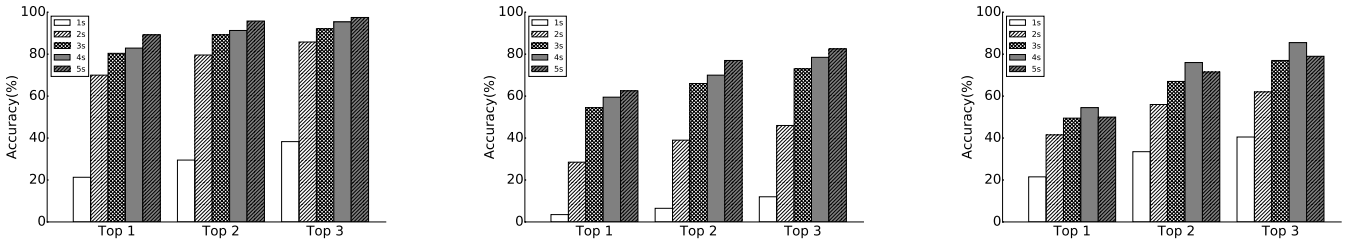
We first tried to classify the foreground apps using single features. In these experiments, we still used our classification framework, but conducted training and testing with each of the 6 features separately (with $l = 1$ in each test). The results are shown in Fig. 3. As shown in the figure, a single feature does not carry enough information to correctly identify a foreground app. Particularly, most features will yield a classification accuracy of less than 25% for top 1 result; the `active_count` feature has the best performance, with slightly over 40% for top 1 accuracy and almost 60% for top 3 accuracy. In contrast, when the 6 features are combined, the classification results (the bar labeled as "All" in Fig. 3) can reach 89.2% for top 1 accuracy, and 97.5% for top 3 accuracy. *These results suggest that the iOS side-channel attack vectors derived from global statistics are not as powerful as the ones we have seen on Android, which typically leak per-app statistic information.* Therefore, successful side-channel attacks on iOS need to combine multiple side-channel attack vectors. It also explains why we needed a new classification framework for conducting side-channel analysis in this paper.

We also studied if all 5000 data points in the side-channel traces are necessary for classification. Particularly, we used the first 1000 data points (corresponding to 1 second of data collection) for both training and testing and show the classification accuracy in Fig. 4a (the bars with "1s" labels). Of course, in these experiments all 6 features were used. Similarly, we also trained and tested with the first 2000, 3000, 4000 data points, and show the results in the same figure (the bars with "2s", "3s", "4s" labels). We can see from the figure that more data points clearly make classification results better. But the first 2000 data points already contain a large amount of information for classifying a foreground app: 70% for top 1 accuracy, 79.6% for top 2 accuracy, and 85.8% for top 3 accuracy. In contrast, using only the first 1000 data points (*i.e.*, 1 second of data collection) is not enough for classifying the apps, with the top 1 accuracy being 21.3%. This is likely because many signature activities of an app's launching procedure happen between 1 second to 2 seconds after the launch begins.

2) Safari Websites: We randomly selected 100 websites from Alexa Top 500 sites on the web (`alexa.com/topsites`) and Moz Top 500 registered domains (`moz.com/top500`). We used *Cycript* to automate the following process: First, the monitoring app is run in the background; second, after 2 seconds, *Safari* is launched to load a blank page; third, after another 3 seconds, the URL bar is filled with the target website's URL and the "enter" button is pressed; Finally, 10 seconds after the website is visited, *Safari* is killed so that the experiment can be restarted. It is worth noting that traces collected when a website is visited from a clean state—newly started *Safari* with a blank page—is not different from when it is visited from the same tab that has already visited another website. To illustrate this point, we show the network traces of launching the *Safari*, and then visiting `yelp.com` (the upper figure of Fig. 5); of visiting `youtube.com` first and then `yelp.com` (the lower figure of Fig. 5). Though the traces collected for `yelp.com` in these two cases are not exactly the same due to noise, they are very similar in the figure (and also in the actual data). Besides, the beginning of the website visit is easy to identify by the monitoring app, since `ibytes` and `obytes` counters increase drastically when it happens.

The monitoring app collects the first 5000 data points (roughly 5 seconds) once large increases of these counters were observed (*i.e.*, when the "enter" button was pressed). We collected 10 traces for each website (with 6 time series in each trace) and the total number of traces was 1000. 8 traces for each website were randomly selected as training data and the rest were used as test data. So the training dataset and test dataset contains 800 and 200 traces, respectively. Fig. 4b shows the result of classification.

When all the 5000 data points are used for classification, the top 1 classification accuracy could reach 68.5% and top 3 accuracy could reach 84.5%. With fewer data points (*e.g.*, 3000 or 4000 data points), the classification accuracy drops only slightly. However, the top 1 classification accuracy drops to 28.5% when only the first 2-second data is used and to 3.5%



(a) Foreground Apps (b) Safari Websites (c) Maps Searches

Fig. 4: Classification results.

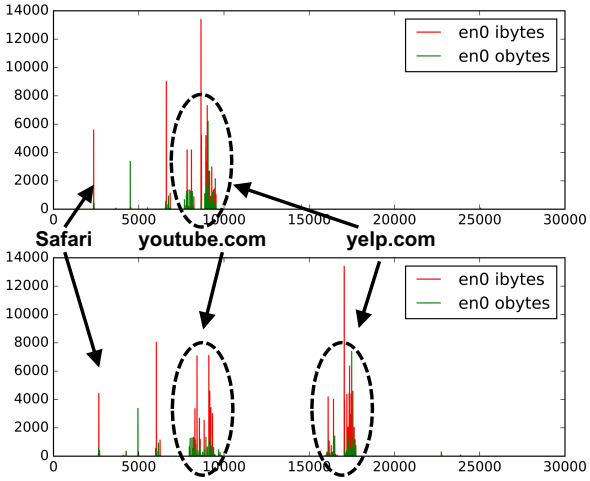


Fig. 5: Network traces of using Safari to visit *Yelp.com* from a clean state, and visit *Yelp.com* after visiting *Youtube.com*.

when only the first 1-second data is used. The results suggest that the website loading typically takes longer than 2 seconds.

3) Apple Maps Searches: We targeted the built-in *Maps* app on iOS in this attack. We selected 100 U.S. National Historic Landmarks from the travel channel (`travelchannel.com`) and Wikipedia. Then we collected 10 traces when searching each of these locations using the *Maps* app. We used *Cycript* to automate the following steps: First, the monitoring app runs in the background; second, the Apple *Maps* is launched after 2 seconds; third, the name of the landmark is filled into the search box and the "enter" button was pressed; finally, 10 seconds later *Maps* is killed. Because the target of the attack is the search location rather than the current location, we did not ask the app to calculate the route. The first 5000 data points (about 5 seconds) after "enter" was pressed were collected as the trace of the corresponding location search. The classification test was performed in the same way as that of the previous two examples. The results of the classification are shown in Fig. 4c. When all the 5000 data points of the 6 features are used, the classifier achieves an accuracy of 50% for top 1 accuracy and 79% for top 3 accuracy. What makes the results interesting is that with only the first 4000 data points, the classification accuracy is slightly better. This is presumably because all signature activities happen within 4 seconds, and the 5th-second data only add noise to the classification experiments. The drop of classification accuracy is significant when only the first 1000 data points are used.

V. Attack 2: Detecting Sensitive In-App Activities

We call the user's specific activities inside an iOS app the in-app activities. In this section, we demonstrate that some in-app activities that lead to severe privacy breach may be identified by sampling memory statistics listed in Table III. We also show how such incidents can be exploited in practical attacks.

Category	Feature
VM	free_count active_count zero_fill_count faults

TABLE III: Attack vectors.

Prior studies such as Zhou *et al.* [67] and Chen *et al.* [32] have demonstrated to use the *exact network* packet sizes and directions to match the patterns of specific user activities in web applications or Android apps. However, we cannot use the same algorithms in our work. We face two new challenges that were not encountered in previous research: (1) iOS do not provide per-app resource usage statistics (*e.g.*, through `procfs` on Android). As a consequence, the side-channel observations are noisy. Therefore, the algorithm, instead of expecting exact matches, must tolerate noise. (2) Any individual attack vector may not manifest clear pattern due to its noisy nature, we need an algorithm to exploit multiple attack vectors at the same time. Therefore, we need to develop a pattern matching algorithms for multi-dimensional data traces.

A. Attack Methods

We developed a pattern matching algorithm that compares two multi-dimensional data traces that are potentially polluted by noise. More formally, given a sample $\vec{X}_t = \{\vec{X}_t^1, \vec{X}_t^2, \cdots, \vec{X}_t^l\}$, where $\vec{X}_t^i = (X_{t_1}^i, X_{t_2}^i, \cdots, X_{t_{n_i}}^i)$, and a signature $\vec{S}_t = \{\vec{S}_t^1, \vec{S}_t^2, \cdots, \vec{S}_t^l\}$, where $\vec{S}_t^i = (S_{t_1}^i, S_{t_2}^i, \cdots, S_{t_{n_i}}^i)$ we aim to measure the distance between the sample and the signature, $d(\vec{X}_t, \vec{S}_t)$.

To address the two technical challenges, background noise and multi-dimensional data (with different length in each dimension), we adopted an extended DTW algorithm [58], which extends the original DTW algorithm to multi-dimensional time series. The multi-dimensional DTW, denoted `DTW_I`, calculates DTW distance for each feature separately, and sums

up each DTW distance after normalization. So the distance between $\vec{X_t}$ and $\vec{S_t}$ would be:

$$d(\vec{X_t}, \vec{S_t}) = \sum_{k=1}^{l} \frac{1}{w_k} \cdot \text{DTW}\big(\vec{X_t^k}, \vec{S_t^k}\big)$$

Here w_k is the normalization factor which is determined empirically. In our attacks, w_k is the average distance between different samples of the signature traces, $\{\vec{S_t}\}$.

The extended DTW algorithm only gives us a relative distance measure: the length of the traces and the level of distortion will both affect the measurement. As such, a pattern matching algorithm using a fixed threshold is likely to be very fragile. In our attacks, to determine whether a sample matches a signature pattern, we compare the sample with multiple signature patterns to which this sample may be related.

B. Case Studies

We collected 10 traces for each in-app activity by sampling the 4 features (in Table III) at the rate of 1000/s. Each trace is comprised of 4 time series. Considering the raw data of any of these time series $\vec{A_t} = \{A_{t_1}, A_{t_2}, \cdots, A_{t_n}\}$, we first calculate the difference between A_{t_i} and A_{t_1}, *i.e.*, $B_{t_i} = A_{t_i} - A_{t_1}$, to construct a new time series $\vec{B_t} = \{B_{t_1}, B_{t_2}, \cdots, B_{t_n}\}$. Then, to reduce the noise in the data, we only keep data points in $\vec{B_t}$ that appear more than 50 times consecutively, and remove the repeated data points. The resulting time series $\vec{X_t}$ is the time series of a feature that we use to calculate the DTW distance.

1) Linking Bitcoin Addresses to iOS Devices: Bitcoin [51] is the most popular cryptographic currency to date. A Bitcoin coin is a chain of ECDSA digital signatures. In each transaction the coin is involved in, the sender signs the hash of the previous transactions of the coin together with the public key of the receiver using her own private key. The receiver can verify the ownership of the coin by verifying the digital signature using the sender's public key. To protect the privacy of the users, the identities of the senders and the receivers in a transaction are replaced by the Bitcoin addresses. A Bitcoin address is the hash value of a public key that the user holds. Each user may possess one or more Bitcoin addresses, thus public/private ECDSA key pairs. Since the coins stored in one address will be spent in their entirety during one transaction, unspent changes will be saved in the original Bitcoin address or, for many Bitcoin wallets, a newly created Bitcoin address.

The target. Anonymity and privacy are desired properties of the Bitcoin network. Neither the payers nor the payees in any transactions should be identified in the public record. Although Bitcoin's strong user privacy is claimed in the original paper of Bitcoin [51], many previous studies have demonstrated that it is still possible to cluster Bitcoin addresses belonging to the same user by conducting transaction graph analysis, and further link these addresses to online merchants because some of their addresses are publicly known [27]. Nevertheless, it is still considered impossible to de-anonymize arbitrary Bitcoin users [49]. The goal of our side-channel analysis is to link the Bitcoin transactions with the monitored mobile user, thus de-anonymizing the transactions that belong to the owner of the iOS device.

Our attack. The high-level idea of our attack is to detect the user's action of making payment with the Bitcoin wallet software on the victim iOS device and record the timestamp of the transaction. Because all successful transactions are included in a public record that can be looked up easily, the adversary is able to correlate the online transaction records with the side-channel detected Bitcoin activity. However, there are a few complications that require us to develop a more polished algorithm than this basic solution. First, because the number of transactions in each block is large (about 2000 as of May 2017 [26]) and the side-channel measured timestamp may be off by seconds, more than one transactions (*e.g.*, in our analysis, up to hundreds) in the online record can be linked to the detected Bitcoin transaction activity. Second, the same Bitcoin address is typically not reused, because many Bitcoin wallets will generate a new Bitcoin address to receive the unspent Bitcoins for each transaction to improve the privacy and anonymity of the users. This artifact makes directly taking the intersection of multiple correlated transaction sets unfeasible. Therefore, we have refined our algorithm as follows.

In order to better describe our attack, we model a Bitcoin transaction, T, as a 3-tuple: $\mathrm{T} = (\mathrm{S}, \mathrm{R}, t)$, where S is the set of payers of the transaction, R is the set of payees of the transaction, and t is the timestamp of the transaction. A payer or payee is one Bitcoin address, a. We particularly use function $\mathrm{T}()$ to represent each element in the tuple. For instance, the timestamp of a transaction is denoted by $\mathrm{T}(t)$ and the payer is $\mathrm{T}(\mathrm{S})$. We also model the i^{th} block in the blockchain, B_i, as a 2-tuple: $\mathrm{B}_i = (\mathbb{T}, t)$. We use $\mathrm{B}_i(\mathbb{T})$ to denote the set of transactions that is associated with block B_i; $\mathrm{B}_i(t)$ to denote the timestamp of the block.

Using method described in Sec. V-A, our monitoring app can detect, and record the timestamp of, each occurrence of the Bitcoin transaction in which the mobile user is the payer. The timestamps of these events are denoted $\{t_1, t_2, t_3, \cdots, t_n\}$, where n is the total number of times that the monitoring app has detected such transactions. We emphasize that the monitoring app does not need to successfully detect all transactions; a non-contiguous subset is usually sufficient. Then for each t_i, $i \in \{1, 2, \cdots, n\}$, we locate the next α blocks in the blockchain; that is, we find a set of α blocks, $\mathbb{B}_i = \{\mathrm{B}_{j+1}, \mathrm{B}_{j+2}, \cdots, \mathrm{B}_{j+\alpha}\}$, so that $\mathrm{B}_{j+1}(t) \geq t_i$ but $\mathrm{B}_j(t) < t_i$. Then we assemble a set of transactions $\mathbb{T}_i = \{\mathrm{T}_k | t_i - \beta < \mathrm{T}_k(t) < t_i + \beta \ \wedge \ \mathrm{T}_k \in \mathbb{B}_i\}$, for each $i \in \{1, 2, \cdots, n\}$. We next exclude any transaction, T_k, from each $\mathbb{T}_i$ ($\mathrm{T}_k \in \mathbb{T}_i$), `iff` $\exists \mathrm{T}_j \in \mathbb{T}_i \ \wedge \ \mathrm{T}_k(S) = \mathrm{T}_j(S) \ \wedge \ k \neq j$, because these transactions are typically from high-profile accounts that reuse their Bitcoin address to perform a large volume of transactions. This heuristic step improves the efficiency of our attack but is not required.

We note that α and β are parameters we can tune. We empirically chose $\alpha = 3$ and $\beta = 15$s. This is because we find typically a transaction will have a very high probability to appear in one of the three following blocks, and 15s is long enough to tolerate the inaccuracy in the transaction timestamps measured by the side-channel analysis.

Then in the list of transaction sets $\{\mathbb{T}_1, \mathbb{T}_2, \cdots, \mathbb{T}_n\}$ that is assembled from the public record, we aim to construct the set of 3-tuples: $X = \{(\mathrm{T}_x, \mathrm{T}_y, \mathrm{T}_z) | \mathrm{T}_x \in \mathbb{T}_i \ \wedge \ \mathrm{T}_y \in \mathbb{T}_j \ \wedge \ \mathrm{T}_z \in \mathbb{T}_k \ \wedge \ \mathrm{T}_x(\mathrm{R}) \cap \mathrm{T}_y(\mathrm{S}) \neq \emptyset \ \wedge \ \mathrm{T}_y(\mathrm{R}) \cap \mathrm{T}_z(\mathrm{S}) \neq \emptyset \ \wedge \ i <$

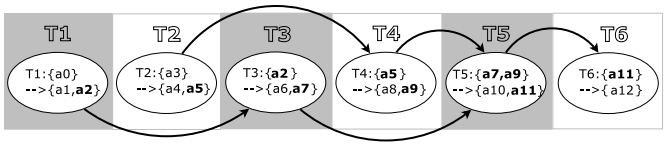


Fig. 6: Blockchain wallet experiment.

$j < k\}$. The transactions in these 3-tuples are initiated from the monitored mobile owner with high probability. Therefore, the Bitcoin addresses that belong to the mobile user is $Y = \{\mathrm{a} | (\mathrm{a} \in \mathrm{T}_x \vee \mathrm{a} \in \mathrm{T}_y \vee \mathrm{a} \in \mathrm{T}_z) \wedge (\mathrm{T}_x, \mathrm{T}_y, \mathrm{T}_z) \in X\}$.

Empirical evaluation. We conducted an empirical attack on the *Blockchain Wallet* iOS app. There are 11 major activities a user can perform in this app[2]. We use our monitoring app to monitor and collect traces from the four `VM` side-channel attack vectors. These activities can be differentiated by observing the patterns in the memory traces. Particularly, for each activity, we collected 10 traces and calculated the average distance between two traces using the extended DTW (see Sec. V-A). The results are shown in Fig. 7a. In this heatmap, each row and each column represent one activity, and cell (i, j) represents the distance between the traces of activity i and activity j. Note the distances are normalized in each *row* so that the average distance between traces of the same activity is 1 and the distances from other activities are normalized accordingly. Therefore, the heatmap is not symmetric.

Besides the targeted activity—the `make-payment` activity—we are also interested in the `return-to-home-screen` activity, because this activity indicates the foreground app is no longer the *Blockchain Wallet* app. This is important because with the capability to detect if the app has been suspended, the adversary only needs to match the signature of an in-app activity with that of other activities in the same app, rather than comparing with all activities in all apps. Fig. 7a clearly shows that the distance between traces in the same activity is much closer than between those of different activities. This is especially true for the `make-payment`(0) activity and `return-to-home-screen`(10) activity. The number in the parentheses is the index in Fig. 7a.

To demonstrate the attack, we created a new Bitcoin account and initially deposited some coins in it. Then we manually made 6 transactions at some random time during 3 days, and at the same time have the monitoring app running in the background to collect the `VM` traces. By faithfully executing the aforementioned steps to collect 6 transaction sets $\{\mathbb{T}_1, \mathbb{T}_2, \mathbb{T}_3, \mathbb{T}_4, \mathbb{T}_5, \mathbb{T}_6\}$ from the public records, we were able to construct the set of 3-tuples $X = \{(\{\mathrm{T}_1, \mathrm{T}_3, \mathrm{T}_5), (\mathrm{T}_2, \mathrm{T}_4, \mathrm{T}_5\}), (\mathrm{T}_3, \mathrm{T}_5, \mathrm{T}_6\}), (\mathrm{T}_4, \mathrm{T}_5, \mathrm{T}_6\})\}$. By linking all transactions in X, a direct acyclic graph is constructed, which is shown in Fig. 6. Therefore the set of Bitcoin addresses that have been used by the user is $\{\mathrm{a}_0, \mathrm{a}_2, \mathrm{a}_3, \mathrm{a}_5, \mathrm{a}_7, \mathrm{a}_9, \mathrm{a}_{11}\}$. Without exception, all of them are correctly identified using the side-channel attacks and the correlation analysis.

2) Other Targets: As shown in the Bitcoin transaction de-anonymization attack example, the binary information leakage via in-app activity detection can be augmented if a public traceable dataset, even though anonymized, is available to the adversary to correlate with the detected event. There are a few other iOS apps that are vulnerable to this type of attacks, such as *Venmo* and *Twitter*.

• *Identify Venmo transactions and user information. Venmo* is a mobile payment service owned by Paypal. It simplifies money transfer processes between banks and accounts. According to a report by Forbes [4], *Venmo* processed about $17.6 billion US Dollars in 2016, which is twice more than the amount in 2015. One interesting aspect of *Venmo*, however, is that by default, all transactions through *Venmo* are shared publicly [9]. Although the users can change it to private, a lot of people do not do so. As of 2014, as many as 50% of all *Venmo* transactions, including their payers, payees, transferred amounts, transfer time and memo, are publicly available [43]. In most cases, the names of payers/payees shown in a transaction are real names. Therefore, by detecting the payment process, it is possible to identify the true identity of the user by matching the transfer time with the public records, as long as the user has not modified the default privacy setting.

In our experiment, similar to the Bitcoin example, we generate the signature of this activity using four `VM` vectors, and run the pattern matching algorithm (Sec. V-A) to detect the activity using our monitoring app. There are 11 major activities in the *Venmo* app that the user can perform, so we collected 10 traces for each activity and computed the average distances between the traces from the same activity and different activities (see Fig. 7b). The distances represented in the heatmap are normalized using the same approach as in Fig. 7a. The first and the last activities are `make-payment`(0) and `return-to-home-screen`(10), respectively. It is clear from the figure that these two activities are easily separable from other activities using the distance measures.

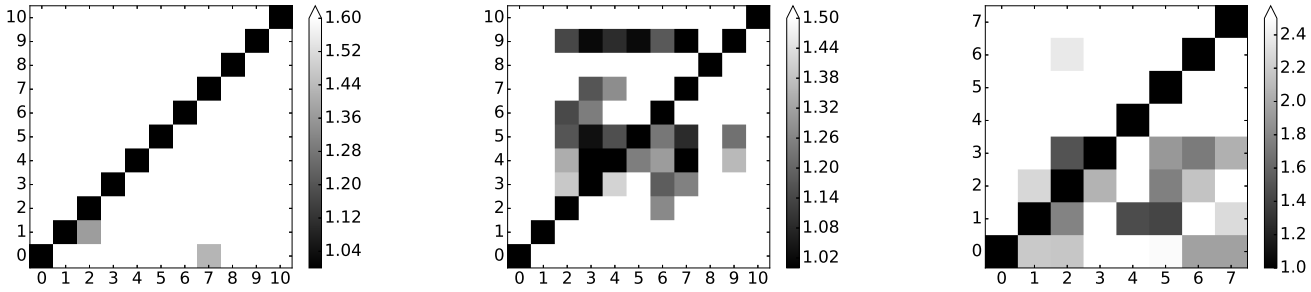
• *Identify Twitter user accounts. Twitter* is one of the most popular social networks in the world. According to a report published in January 2017 [18], *Twitter* has 317 million monthly active users, and there are 500 million tweets being sent every day. Similar to the attack scenario identified by Zhou *et al.* in their study of Android side channels [67], if an adversary is able to identify the user's action of posting tweets in the *Twitter* app (using `VM` features), by correlating with the online database of tweets, the user's identity can be identified through such side-channel analysis.

In our experiment, we have shown that the user tweet-posting activity has a unique and stable `VM` signature. Using our monitoring app and the algorithm mentioned in Sec. V-A, we can reliably detect the time of the `post-tweets`(0) activity and `return-to-home-screen`(7) activity (the first and last activities in Fig. 7c). Therefore, it is also possible to identify the user's real identity. As of the time of writing, the public available tweeting record is no longer complete. Nevertheless, partially released dataset still allow correlation analysis.

VI. Attack 3: Bypassing Sandbox Restrictions

As mentioned in Sec. III, the `fileExistsAtPath` API can be used to check whether a file or directory exists even without proper permission. In this section, we show how

[2] `make-payment`(0), `menu-addresses`(1), `menu-backupfunds`(2), `menu-merchantmap`(3), `menu-settings`(7), `menu-support`(8), `send-button`(6), `receive-button`(4), `overview-button`(9), `scan-QR-code`(5), and `return-to-home-screen`(10).



(a) Blockchain Wallet (b) Venmo (c) Twitter

Fig. 7: In-app activity heatmaps.

a malicious iOS app can leverage this API to bypass iOS sandbox restrictions to detect the existence of cross-container files and extract sensitive user information.

Depth	File Path
4	/private/var/logs/lockdownd.log
5	/private/var/logs/AppleSupport/general.log
6	/private/var/mobile/Library/DataAccess/AccountInformation.plist
7	/private/var/mobile/Library/Spotlight/BundleInfo/InstalledApps.plist
8	/private/var/mobile/Library/Caches/com.apple.purplebuddy/ com.apple.opengl/linkCache.data

TABLE IV: Absolute file paths for Fig. 8.

A. Attack Methods

We further empirically evaluated the characteristics of the `fileExistsAtPath` timing channel. Specifically, we measured the execution time of the `fileExistsAtPath` API using `mach_absolute_time()` on a non-jailbroken iphone7 device running iOS 10.2.1, while varying the depths of the input file paths. The tested file paths are listed in Table IV. The iOS app making the API calls did not have the permission to access these files, so the query to the `fileExistsAtPath` API all failed. For each file path, we run 100 trials. In each trial, we queried the API 50 times and calculated the mean value of the execution time. We compute the mean and standard deviation of these 100 trials and plot them in Fig. 8. Also shown in the figure are the execution times of the API with inputs of some non-existent files at the same depth as the tested file paths. From Fig. 8, we can see that the two cases (*i.e.*, existent vs. non-existent) are clearly distinguishable. Note the unit of the `y`-axis of the figure is the *Mach Absolute Time Unit*, which is a value returned by `mach_absolute_time()`. This unit time is CPU-dependent, which can be converted to nanoseconds using a system-provided API [17].

As such, to determine the existence of a file or directory outside an app's sandbox, a timing channel can be reliably constructed by the app using the differential tests that follow: It first queries the `fileExistsAtPath` API with the targeted file or directory as input 50 times and measures the average execution time. Then it compares the average execution time with that for a non-existent file (*e.g.*, a file with a random string in its name) of the same depth in the filesystem. In this way, the timing difference will reliably tell whether the file exists or not.

B. Case Studies

By utilizing the `fileExistsAtPath` timing channels, a malicious iOS app could bypass the sandbox restrictions enforced by iOS and learn information about other apps. Specifically, we show that the technique can be used to infer that the list of apps installed on the device (Sec. VI-B1), as well as information regarding photos, videos, and voice memos (Sec. VI-B2). In each of the attacks we show, we use the differential analysis technique outlined in Sec. VI-A to determine the existence of the specific files on a non-jailbroken iphone7 with iOS 10.2.1. In all tests, we can reliably detect the file existence without *any* false detection.

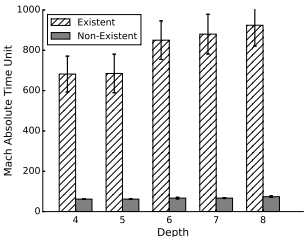


Fig. 8: Execution time of `FileExistAtPath` when changing the depth.

1) List of installed apps: Apps that a user has installed on the phone sometimes reveal the user's life styles or personal choices of vendors (*e.g.*, *Verizon*, *Marriott*, *Chase*, *Hertz*, *United*, *etc*). These apps can be used to profile the user of the device, which can be valuable to ads providers. Nevertheless, the existence of certain apps may leak more sensitive information about the user. For instance, the fact that a user has installed *Hornet*, a same-sex-dating app, may reveal the user's sex orientation; and *Pregnancy+*, as its name indicates, may suggest that the user is trying to get pregnant or already pregnant. We list some of these sensitive apps in Table V.

To learn what apps have been installed on the same device, one approach is to use a private API in the `LSApplicationWorkspace` class, which provides a list of installed apps [11]. However, the use of this private API will be detected in the vetting process and result in rejection of the app. Another method is through the `canOpenURL` API (*i.e.*, `[UIApplication canOpenURL:]`), which allows an app to check whether there is another app to handle a certain URL. However, this API was extensively misused by developers to obtain the installed app list. As a response, since iOS 9, Apple has imposed limits on the use of this API by requiring explicit declaration of all the targeted schemes in the `plist` file [2].

Despite Apple's effort of stronger cross-app isolation, we found it is still possible for third-party apps to stealthily query whether a certain app has been installed using the `fileExistsAtPath` timing channel, if one of the two following conditions is met for this targeted app: (1) it requires permission for sending push notifications, or (2) it dynamically registers home screen quick actions.

• *Push notifications.* When an iOS app that requires sending push notifications is launched for the first time, it will prompt the user for permission to send notifications. On iOS 9, no matter whether the user grants the permission or not, a `.pushstore` file with the bundle identifier as its name will be created in a specific directory (*e.g.*, `/var/mobile/Library/SpringBoard/PushStore/com.google.Gmail.pushstore` for the *Gmail* app). On iOS 10, this file will be created the first time the app receives a push notification. A large portion of iOS apps request push notification. Particularly, we conducted a static analysis on the dataset of 7,418 apps, and found that 4,980 (67.13%) of them *may* use local notification to alert user about in-app information and 4,438 (59.83%) *may* use remote notification to handle messages pushed from a remote server. The union of the two sets are 5,886 (79.35%) apps. We further randomly installed 150 apps from iOS App Store and found that push notification was requested by 67 (*i.e.*, 44.7%) of these apps.

• *Home screen quick actions.* The feature of home screen quick actions was first introduced in iOS 9 with 3D-touch enabled iphone 6S and iphone 6S plus in September 2015 [1], [10]. It allows users to have quick accesses to certain functionalities of an app by long-pressing the icon on the home screen, without opening the app. There are two ways to register quick actions: static (by defining them in `Info.plist`) or dynamic (by defining them in code). Through our experiment, we found that when an app which registers home screen quick actions dynamically is installed and launched for the first time, a new `.plist` file will be created in a specific directory with its bundle identifier as the name of the file (*e.g.*, `/var/mobile/Library/SpringBoard/ApplicationShortcuts/com.google.Gmail.plist` for the *Gmail* app). Though this new iOS feature brings convenience to users, it also introduces a vector for information leakage. We can identify the installed apps by detecting whether a specific `plist` file has been created. We studied 150 top ranked apps (*i.e.*, top 150 apps in App Store's "Top Charts"), and 47 of them (31.33%) have dynamically registered quick actions.

We conducted a measurement study of the aforementioned techniques, and discovered that most of the sensitive apps we examined can be detected using at least one of these approaches. In Table V, we show the results of 8 examples, which reveal different aspects of a user, including sexual orientation (*Hornet*), health condition (*AsthmaMD*), age and education (*Ready4SAT*), marital status (*DivorceForce*), drug condition (*Weedmaps*), *etc*.

2) Other attack targets: The `fileExistsAtPath` timing channel is capable of extracting other information. For example, iOS stores photos and video clips in fixed paths and predictable names (*i.e.*, `<num>APPLE/IMG_<index>.<ext>`, where `<num>` is an integer starting from 100 and `<index>` starts from 0001). Enumerating all possible combinations only needs a few seconds. Also, the pre-installed *Voice Memos* app records speeches and conversations and names the recordings using `<timestamp>`, which is the time of recording that is accurate to a second. In our experiment, we found that enumerating each second of one year (*e.g.*, 2016) only take 80 minutes on an iPhone 7. Therefore, the same timing channel can be used to learn the number of photos/videos/memos of the user, and also infer the timestamps that memos were taken.

App Name	Description	A	B
Hzone	HIV dating app for HIV positive singles	✓	✗
AsthmaMD	Asthma activity tracking & visualization	✓	✗
Hornet	Social network for gay men	✓	✗
Pregnancy+	Pregnancy & baby developing tracker	✓	✗
Sugar Sense	Diabetes app to track blood sugar level	✓	✓
Weedmaps	Marijuana directory and discovery source	✓	✗
Ready4SAT	Preparation for the SAT test	✓	✓
DivorceForce	Community of those affected by divorce	✓	✗

TABLE V: Examples of sensitive apps. A: requesting push notifications; B: dynamically registering quick actions.

VII. Practical Issues

In this section, we discuss several issues in practical side-channel attacks, and how we addressed them in our work.

A. Run Background Apps on iOS

To be able to run in the background, an app needs to specify one of the nine "Background Modes" [5] (*e.g.*, `Audio`, `VoIP`, `Location updates`, *etc*) in its `Info.plist` file. When it is launched for the first time, it will explicitly ask for the user's permission to run in the background. Some background-mode permissions (*e.g.*, `Location updates`) will need the user to explicitly grant permission every now and then.

To periodically (in our experiments every 1ms) invoke our monitoring thread in the background with fixed intervals, we used the `NSTimer` class, which can be used to create a timer object that expires after a certain time interval has elapsed, and sends a specified message to a target object [13]. In particular, we used `[NSTimer scheduledTimerWithTimeInterval: target: selector: userInfo: repeats:]` API to schedule the timer to execute our monitoring process in a fixed interval. Our experiments show that with an `Audio` background permission, our monitoring app can keep running and periodically invoke monitoring thread in the background to sample the features.

B. App Store Vetting

We submitted a monitoring app that is able to conduct all the aforementioned side-channel attacks to the App Store for vetting. The app is disguised as an *Audio Player*, which requires the `Audio` Background Mode for running in the background. The app collects all the 6 features from Table I. The sampling rate of each feature is about 1000/s. We also included the code for conducting `fileExistsAtPath` timing channel attacks in the monitoring app. Our app successfully passed the vetting, which indicates that these codes are not considered as malicious by Apple. After our app was approved by Apple, we downloaded the app and withdrew it immediately.

C. Background Noise

Intuitively, because most of our attacks exploit global statistic information of the system resources, our attacks may be fragile. However, in the presented attacks, we did not intentionally clean up the background apps, but the noise in our collected traces remains manageable using our machine learning frameworks. This is because iOS itself suspends apps when they run in the background, unless the app specially requests `background` permissions. Therefore, iOS devices are relatively quieter than Android devices, which greatly facilitates side-channel attacks.

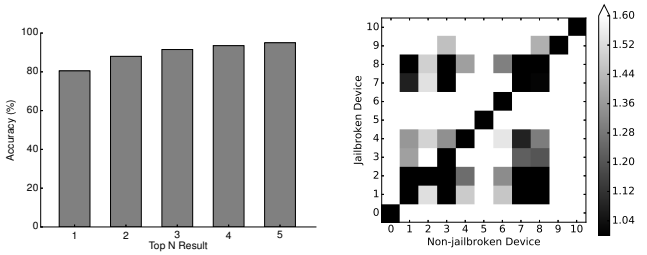
D. Cross-device Attack Feasibility

To demonstrate the practicality of our attacks, we manually collected traces on another non-jailbroken iPhone 7 running iOS 10.2.1. Then, we use previously collected data as training set (for classification) or signature (for pattern matching) to re-evaluate the attacks. To make sure the sampling rate on different devices and different iOS versions remain the same (about 1000 times/s), our monitoring app self-adjusts the sampling interval (specified using `NSTimer`) the first time it runs in the background.

1) Classification: We randomly selected 20 third-party apps from the 100 apps we used in Sec. IV. Then we manually collected 10 traces of the app launching process for each of them to construct a new test set, so this new test set contains 200 traces. Then, we used previous training set as mentioned in Sec. IV and repeated the evaluation. Fig. 9a shows that the performance of the classifier drops only slightly: 80.5% accuracy with Top 1 result, 91.5% accuracy for Top 3, and 95.0% for Top 5. It is worth noting that the training traces were collected *20 days before* the new test set on a *different* device with a *different* iOS version. Moreover, some of the 20 apps have been updated during the time period. For instance, *Blockchain Wallet* has been updated, but we are still able to detect the launching process of it with high confidence (90% Top 1 accuracy for this app).

2) Pattern Matching: We recollected traces of 11 major activities in *Blockchain Wallet*, 5 traces each. We randomly selected 5 previously-collected traces of each activity (Sec. V-B1) as the signature traces, and used the similar method to draw a heatmap (Fig. 9b). This time, x-axis means non-jailbroken (testing) device, and y-axis means jailbroken (training) device. We normalized the distance per row using $cell(i, j)$ as the base (w_k in Sec. V-A), so that each diagonal value is 1. From Fig. 9b, we can clearly see that the `make-payment`(0) and `return-to-home-screen`(10) and activity can be clearly distinguished. Some activities are not so distinguishable (*e.g.*, `menu-settings`(7) and `menu-backupfunds`(2)) because these activities consist of common sub-activities (*i.e.*, clicking the menu button).

From these experiments, we show that our demonstrated attacks are robust enough to be trained on a device owned by the attacker and then tested using the data collected from the victim's device. Minor differences in the iOS versions and app versions can also be tolerated.



(a) Classification. (b) Pattern matching.

Fig. 9: Cross-device evaluation.

E. Power Consumption

We evaluated the power consumption of the monitoring app when collecting side-channel data. Specifically, we ran the experiment on a jailbroken iphone7 with iOS 10.1.1. The phone was fully charged before the experiment. It ran in the foreground and read the APIs 1000 times per second. This setup over-approximates its power consumption when it runs in the background. The monitoring app called `[UIDevice batteryLevel]` every 1 minute to keep track of the remaining battery level. After 60 minutes, only 5% of battery was consumed, *i.e.*, less than 1% per 10 minutes. The experiment suggests that the monitoring app will not drain the battery much faster than regular apps.

F. Execution Time of Machine Learning Algorithms

We evaluated the execution time of these algorithms with the experiment data used in Sec. IV-B1. Our original data contains 960 training samples; each sample has 6 time series; and each time series consists of 5000 data points. To compare the performance of these two algorithms, we report in Table VI the execution time of classifying one test trace using these two algorithms when each time series has 50, 500, 5000 data points, and when the training set is composed of 480 or 960 training samples. Note that the kNN algorithm does not have a training phase. It compares each testing sample with all training samples to calculate the distance. From the table, we can see the execution time of kNN algorithm is linear to the number of training samples. The SVM algorithm has separated training and testing phases. It is the execution time of the testing phase that is of interest (reported in Table VI). The execution time of SAX+BOP+SVM algorithm is significantly shorter than kNN in our case; when the trace has 5000 data points, the kNN algorithm took too long to complete. Therefore, the classification framework presented in Sec. IV-A is much more efficient.

In contrast, the reason for selecting kNN+DTW algorithm for the pattern matching task in Sec. V is that the number of in-app activities for each app is small (*e.g.*, 11 in both *Blockchain wallet* and *Venmo*) and the number of data points are fewer (*e.g.*, on average 30.54 data points in our experiments). To compute the distance using the multi-dimensional DTW algorithm (with the 4 VM features) between two traces, the average execution time was only 0.03 second.

data points	50		500		5000	
classifier	kNN	SVM	kNN	SVM	kNN	SVM
480 training samples	48	0.07	536	0.11	—	0.15
960 training samples	94	0.08	1095	0.13	—	0.17

TABLE VI: Execution time comparison (in seconds).

VIII. Countermeasures

We formulated the following countermeasures and discussed them with a group of Apple engineers who were assembled specifically to address our attacks in iOS.

• *Eliminating the attack vectors.* Removing the APIs that lead to information leakage will completely eliminate the threats. However, as stated in Sec. III, among the 7,418 iOS apps we statically analyzed, `host_statistics64()` is probably used by about 1,230 apps and `getifaddrs()` is potentially used by 3,955 apps. Eliminating such widely used APIs may cause significant compatibility issues. This concern has also been confirmed by Apple engineers, who believed it is difficult to simply remove these APIs from iOS because they are used by some high-profile apps.

• *Rate limiting.* In our (first two categories of) attacks, the monitoring app calls the APIs at a rate of 1000 times per second. It is intuitive that by limiting the rate at which an app can query the sensitive APIs, most applications may still work while the attacks can be mitigated. We envision rating limiting can be implemented by caching the return values in the kernel and updating the cached value only N times per second. Then in our experiments, every $(1000/N)$th data point of our original data is preserved. To evaluate the effectiveness of these methods, we filtered data points accordingly for both training and testing, and then repeated the experiments in Sec. IV-B1. The top 1, 2, and 3 classification accuracy with the maximum sample rate of 5, 10, 100, 500, and 1000 per second are shown in Fig. 10a. From the figure we can see that a sample rate of 10 per second is still high enough to conduct side-channel attacks, with top 1 accuracy of 69.6% and top 3 accuracy of 78.3%. This result also suggest our classification framework remains robust even with less data. The effectiveness of the attacks decreases dramatically when the sample rate drops to $5/s$, however. We have discussed these ideas and results with Apple and were informed that rate limiting has been implemented in iOS 11.1 for `host_statistics64()`, as well as macOS High Sierra 10.13.1, watchOS 4.1 and tvOS 11.1.

• *Coarse-grained return values.* Another approach to countering the attacks is to reduce the granularity of the return values. For instance, instead of returning the exact number of page faults, free pages, or bytes sent/received from the $Wifi$ interface, *etc*., the last 1, 2, or 3 decimal digits of the values can be masked. We evaluated this method for the experiment we did in Sec. IV-B1 by reducing the granularity of all 6 features. We show the top 1, 2, and 3 accuracy of the classification in Fig. 10b. As seen in the figure, when reducing the granularity of the return values, the classification accuracy decreases accordingly. The accuracy drops to a reasonably low level when masking 3 digits (28.8% top 1 accuracy and 41.3% top 3 accuracy). Apple has implemented this approach for `getifaddrs()` in iOS 11 to round the values of `ibytes` and `obytes` to `1K Bytes`.

• *Runtime detection.* An alternative approach is to monitor the use of the leaky APIs while some sensitive apps are running in the foreground. This idea has been illustrated in Android by Zhang *et al.* [65] on Android using a non-privileged *guardian* app. Due to the more strict cross-app isolation on iOS, however, this task can only be accomplished by the system itself on iOS.

• *Privacy-preserving statistics reporting.* Xiao *et al.* [61] proposed a privacy-preserving `procfs` to mitigate side channels resulted from the `procfs` in Linux OS, so that statistics reporting through `procfs` satisfies *d-privacy*, a variation of differential privacy. Apple could modify the OS kernel and implement similar functionalities to these leaky APIs. The effectiveness and performance overhead of such approach on iOS warrant further research.

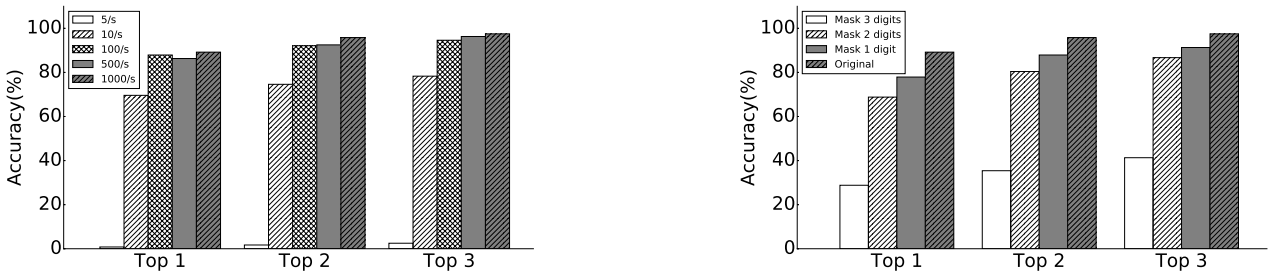
• *Removing the `fileExistsAtPath` timing channel.* Apple has made kernel-level changes in its VFS implementation to eliminate this timing channel. We have confirmed that the timing channel has been eliminated in iOS 11.

IX. Related Work

Closest to our work is the studies of `procfs` side channels on Linux and Android. Particularly, Zhang and Wang [64] demonstrated side-channel attacks through `procfs` on Linux to eavesdrop users' keystrokes. Jana and Shmatikov [42] exploited `procfs` on Linux to infer the website a Chrome browser visits by taking snapshots of its memory footprint (*e.g.*, data resident size). Qian *et al.* [56] exploited error packet counters reported in `/proc/net/` to facilitate off-path TCP session hijacking attacks. Zhou *et al.* [67] demonstrated inference attacks using `procfs` on Android to learn a victim app's activity by learning its packet statistics. Chen *et al.* [31] extracted the victim app's CPU utilization time, memory usage, and network usage from various `procfs` files to detect activity transition and then identify the foreground activity. Lin *et al.* [45] employed `procfs` to extract an app's CPU usage to detect user's key press operation on Android. Zhang *et al.* [65] explored similar channels from `procfs` to fingerprint user behavior through the Android apps of IP cameras. Most recently, Diao *et al.* [36] studied the use of global interrupt counters in `procfs` to infer the user's unlock patterns and foreground apps.

Some other research has explored the use of mobile sensors for constructing side channels. Besides location leakage through GPS [44], [55], accelerometers [23], [41], [47], [53], [59], magnetometer [57], gyroscope [50], [52], [59], and orientation sensor [29], [63] have also been exploited to infer the user's location, movement, and even keystrokes (thus PIN and passwords). These papers all studied sensor-based side channels on Android. Unlike Android which allows a third-party app to stealthily use these sensors, iOS requires special entitlements to use these sensors. For instance, to acquire GPS information, an iOS app needs to ask the user to authorize GPS uses explicitly [7]; to use motion sensors, such as accelerometers, magnetometers, and gyroscopes, starting from iOS 10, developers must place `NSMotionUsageDescription` into `Info.plist` [19]. However, once the permission is granted, similar side-channel attacks may be conducted on iOS devices.

There were only a few past work exploring iOS side channels. But their threat models were very different from



(a) Rate limiting.

(b) Coarse-grained return values.

Fig. 10: Countermeasure experiments on app classification.

ours. For example, Marquardt *et al.* utilized accelerometers on iPhone 4 to perform inference attack against a keyboard placed next to the device [48], while our work targets at other apps on the same device. Genkin *et al.* [39] demonstrated that using magnetic probes placed close to the iPhone or power probes connected to the iPhone's USB cable, ECDSA keys used in OpenSSL and CoreBitcoin on iPhones can be extracted. Our attacks do not assume physical possession of the device by the attacker. Therefore, magnetic or power attacks are out of the scope of our threat model.

Some existing studies focus on iOS security, but not on side-channel leakage. For example, Wang *et al.* [60] proposed a method to inject exploitable vulnerabilities in iOS app to bypass the app vetting. Xing *et al.* [62] discovered a series of flaws in iOS and OS X, which allow the attacker to gain unauthorized access to other apps' sensitive data. Deshotels *et al.* [35] examined the flaws in iOS sandbox profiles and showed how an app can utilize them to learn sensitive information about the user.

X. Conclusion

In this paper, we presented the first exploration of OS-level side channels on iOS. Our study suggests that although iOS does not have `procfs` or permit querying per-app statistic information, there are still APIs that allow a third-party app to query global statistics of the memory and network resources, or to construct timing channels to break filesystem sandboxes. We show three categories of side-channel attacks that exploit these APIs to extract private user information, which include inferring foreground apps, fingerprinting visited websites, identifying map searches, de-anonymizing users of *Bitcoin Wallet*, detecting installed apps, *etc*. These demonstrated attacks showed that similar to Android, cross-app side-channel attacks on iOS are also feasible. Our study has helped Apple mitigate these security threats in iOS/MacOS.

Acknowledgements

We thank the Apple engineers who diligently worked on adjusting iOS/MacOS to address the security issues we discovered in this paper. We also thank the anonymous reviewers for their valuable comments. The project is supported in part by NSF grant 1718084, 1566444, 1527141, 1618493, ARO W911NF1610127 and a Samsung gift fund.

References

[1] "Adopting 3d touch on iphone: Getting started with 3d touch." https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/Adopting3DTouchOniPhone/.

[2] "Api reference: canopenurl(:)," https://developer.apple.com/reference/uikit/uiapplication/1622952-canopenurl.

[3] "App sandbox in depth," https://developer.apple.com/library/content/documentation/Security/Conceptual/AppSandboxDesignGuide/AppSandboxInDepth/AppSandboxInDepth.html.

[4] "As venmo's popularity explodes, its customer service team scrambles to keep up - forbes." https://www.forbes.com/sites/laurengensler/2017/02/14/venmo-customer-service/\#5e00fd081cfd.

[5] "Background execution," https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/BackgroundExecution/BackgroundExecution.html.

[6] "Capstone," http://www.capstone-engine.org.

[7] "Cllocationmanager-api reference," https://developer.apple.com/reference/corelocation/cllocationmanager.

[8] "Cycript - jay freeman(saurik)." http://www.cycript.org.

[9] "Helpful information - venmo." https://venmo.com/legal/us-helpful-information/.

[10] "Home screen actions - extensions - ios human interface guidelines." https://developer.apple.com/ios/human-interface-guidelines/extensions/home-screen-actions/.

[11] "ios-runtime-headers," https://github.com/nst/iOS-Runtime-Headers/blob/master/Frameworks/MobileCoreServices.framework/LSApplicationWorkspace.h.

[12] "Libsvm - faq." http://www.csie.ntu.edu.tw/~cjlin/libsvm/faq.html.

[13] "NSTimer," https://developer.apple.com/reference/foundation/nstimer.

[14] "Piecewise aggregate approximation (paa) | sax-vsm." https://jmotif.github.io/sax-vsm_site/morea/algorithm/PAA.html.

[15] "/proc on mac os x - mac os x internals." http://osxbook.com/book/bonus/ancient/procfs.

[16] "Security and privacy changes in ios 9 | in security." https://nabla-c0d3.github.io/blog/2015/06/16/ios9-security-privacy/.

[17] "Technical q&a qa1398: Mach absolute time units - apple developer." https://developer.apple.com/library/content/qa/qa1398/_index.html.

[18] "Twitter by the numbers (2017): Stats, demographics & fun facts." https://www.omnicoreagency.com/twitter-statistics.

[19] "Working with security and privacy," https://developer.xamarin.com/guides/ios/application_fundamentals/security-privacy-enhancements/offline.pdf.

[20] "Z-normalization | sax-vsm." https://jmotif.github.io/sax-vsm_site/morea/algorithm/znorm.html.

[21] Apple, "App store review guidelines," https://developer.apple.com/app-store/review/guidelines/.

[22] ——, "Mach overview," https://developer.apple.com/library/content/documentation/Darwin/Conceptual/KernelProgramming/Mach/Mach.html.

[23] A. J. Aviv, B. Sapp, M. Blaze, and J. M. Smith, "Practicality of accelerometer side channels on smartphones," in *the 28th Annual Computer Security Applications Conference*. ACM, 2012.

[24] P. Belgarric, P.-A. Fouque, G. Macario-Rat, and M. Tibouchi, "Side-channel analysis of weierstrass and koblitz curve ecdsa on android smartphones," in *Cryptographers' Track at the RSA Conference*. Springer, 2016.

[25] D. J. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series." in *KDD workshop*. Seattle, WA, 1994.

[26] blockchain.info, "Average number of transactions per block," https://blockchain.info/charts/n-transactions-per-block.

[27] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten, "Sok: Research perspectives and challenges for bitcoin and cryptocurrencies," in *IEEE Symposium on Security and Privacy*, 2015.

[28] J. Brownlee, "Apple app store now rejecting app code for private api calls," http://www.geek.com/apple/apple-app-store-now-rejecting-app-code-for-private-api-calls-983411/.

[29] L. Cai and H. Chen, "Touchlogger: Inferring keystrokes on touch screen from smartphone motion." *HotSec*, 2011.

[30] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, 2011, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[31] 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 *23th USENIX Security Symposium*, 2014.

[32] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: A reality today, a challenge tomorrow," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010.

[33] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE transactions on information theory*, 1967.

[34] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu, "iris: Vetting private api abuse in ios applications," in *the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[35] L. Deshotels, R. Deaconescu, M. Chiroiu, L. Davi, W. Enck, and A.-R. Sadeghi, "Sandscout: Automatic detection of flaws in ios sandbox profiles," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[36] W. Diao, X. Liu, Z. Li, and K. Zhang, "No pardon for the interruption: New inference attacks on android through interrupt timing analysis," in *37th IEEE Symposium on Security and Privacy*, 2016.

[37] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications." in *Network and Distributed System Security Symposium*, 2011.

[38] T. F. Foundatio, "The freebsd project," https://www.freebsd.org.

[39] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, "Ecdsa key extraction from mobile devices via nonintrusive physical side channels," in *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[40] G. Goller and G. Sigl, "Side channel attacks on smartphones and embedded devices using standard radio equipment," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2015.

[41] J. Han, E. Owusu, L. T. Nguyen, A. Perrig, and J. Zhang, "Accomplice: Location inference using accelerometers on smartphones," in *2012 Fourth International Conference on Communication Systems and Networks*. IEEE, 2012.

[42] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," in *2012 IEEE Symposium on Security and Privacy*, 2012.

[43] B. Kraft, E. Mannes, and J. Moldow, "Security research of a social payment app," 2014.

[44] M. Li, H. Zhu, Z. Gao, S. Chen, L. Yu, S. Hu, and K. Ren, "All your location are belong to us: Breaking mobile social networks for automated user location tracking," in *the 15th ACM international symposium on Mobile ad hoc networking and computing*. ACM, 2014.

[45] C.-C. Lin, H. Li, X. Zhou, and X. Wang, "Screenmilker: How to milk your Android screen for secrets," in *21st ISOC Network and Distributed System Security Symposium*, 2014.

[46] J. Lin and Y. Li, "Finding structural similarity in time series data using bag-of-patterns representation," in *International Conference on Scientific and Statistical Database Management*. Springer, 2009.

[47] X. Liu, Z. Zhou, W. Diao, Z. Li, and K. Zhang, "When good becomes evil: Keystroke inference with smartwatch," in *the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[48] P. Marquardt, A. Verma, H. Carter, and P. Traynor, "(sp) iphone: decoding vibrations from nearby keyboards using mobile phone accelerometers," in *the 18th ACM conference on Computer and communications security*. ACM, 2011.

[49] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage, "A fistful of bitcoins: Characterizing payments among men with no names," in *Internet Measurement Conference*. ACM, 2013.

[50] Y. Michalevsky, D. Boneh, and G. Nakibly, "Gyrophone: Recognizing speech from gyroscope signals," in *23rd USENIX Security Symposium*, 2014.

[51] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," https://bitcoin.org/bitcoin.pdf.

[52] S. Nawaz and C. Mascolo, "Mining users' significant driving routes with low-power sensors," in *the 12th ACM Conference on Embedded Network Sensor Systems*. ACM, 2014.

[53] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang, "Accessory: password inference using accelerometers on smartphones," in *the 12th Workshop on Mobile Computing Systems & Applications*. ACM, 2012.

[54] P. Patel, E. Keogh, J. Lin, and S. Lonardi, "Mining motifs in massive time series databases," in *2002 IEEE International Conference on Data Mining*. IEEE, 2002.

[55] I. Polakis, G. Argyros, T. Petsios, S. Sivakorn, and A. D. Keromytis, "Where's wally?: Precise user discovery attacks in location proximity services," in *the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[56] Z. Qian, Z. M. Mao, and Y. Xie, "Collaborative TCP sequence number inference attack: How to crack sequence number under a second," in *19th ACM Conference on Computer and Communications Security*, 2012.

[57] C. Shen, S. Pei, T. Yu, and X. Guan, "On motion sensors as source for user input inference in smartphones," in *IEEE International Conference on Identity, Security and Behavior Analysis*. IEEE, 2015.

[58] M. Shokoohi-Yekta, J. Wang, and E. Keogh, "On the non-trivial generalization of dynamic time warping to the multi-dimensional case," in *the 2015 SIAM International Conference on Data Mining*. SIAM, 2015.

[59] H. Wang, T. T.-T. Lai, and R. Roy Choudhury, "Mole: Motion leaks through smartwatch sensors," in *the 21st Annual International Conference on Mobile Computing and Networking*. ACM, 2015.

[60] T. Wang, K. Lu, L. Lu, S. P. Chung, and W. Lee, "Jekyll on ios: When benign apps become evil." in *USENIX Security Symposium*, 2013.

[61] Q. Xiao, M. K. Reiter, and Y. Zhang, "Mitigating storage side channels using statistical privacy mechanisms," in *22nd ACM Conference on Computer and Communications Security*, 2015.

[62] L. Xing, X. Bai, T. Li, X. Wang, K. Chen, X. Liao, S.-M. Hu, and X. Han, "Cracking app isolation on apple: Unauthorized cross-app resource access on mac os," in *the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[63] Z. Xu, K. Bai, and S. Zhu, "Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors," in *the 5th ACM conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2012.

[64] K. Zhang and X. Wang, "Peeping Tom in the neighborhood: Keystroke eavesdropping on multi-user systems," in *18th USENIX Security Symposium*, 2009.

[65] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang, "Leave me alone: App-level protection against runtime information gathering on android," in *36th IEEE Symposium on Security and Privacy*, 2015.

[66] X. Zhang, Y. Xiao, and Y. Zhang, "Return-oriented flush-reload side channels on ARM and their implications for android devices," in *SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[67] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt, "Identity, location, disease and more: Inferring your secrets from Android public resources," in *20th ACM Conference on Computer and Communications Security*, 2013.