The Danger of Minimum Exposures: Understanding Cross-App Information Leaks on iOS through Multi-Side-Channel Learning

Zihao Wang*
Indiana University Bloomington
zwa2@iu.edu

Jiale Guan*
Indiana University Bloomington
guanjia@iu.edu

XiaoFeng Wang
Indiana University Bloomington
xw7@indiana.edu

Wenhao Wang
Institute of Information Engineering, Chinese Academy of Sciences
wangwenhao@iie.ac.cn

Luyi Xing
Indiana University Bloomington
luyixing@indiana.edu

Fares Alharbi
Indiana University Bloomington
fafaalha@iu.edu

ABSTRACT

Research on side-channel leaks has long been focusing on the information exposure from a single channel (memory, network traffic, power, etc.). Less studied is the risk of learning from multiple side channels related to a target activity (e.g., website visits) even when individual channels are not informative enough for an effective attack. Although the prior research made the first step on this direction, inferring the operations of foreground apps on iOS from a set of global statistics, still less clear are how to determine the maximum information leaks from all target-related side channels on a system, what can be learnt about the target from such leaks and most importantly, how to control information leaks from the whole system, not just from an individual channel.

To answer these fundamental questions, we performed the first systematic study on multi-channel inference, focusing on iOS as the first step. Our research is based upon a novel attack technique, called Mischief, which given a set of potential side channels related to a target activity (e.g., foreground apps), utilizes probabilistic search to approximate an optimal subset of the channels exposing most information, as measured by Merit Score, a metric for correlation-based feature selection. On such an optimal subset, an inference attack is modeled as a multivariate time series classification problem, so the state-of-the-art deep-learning based solution, InteceptionTime in particular, can be applied to achieve the best possible outcome. Mischief is found to work effectively on today's iOS (16.2), identifying foreground apps, website visits, sensitive IoT operations (e.g., opening the door) with a high confidence, even in an open-world scenario, which demonstrates that the protection Apple puts in place against the known attack is inadequate. Also importantly, this new understanding enables us to develop more comprehensive protection, which could elevate today's side-channel research from suppressing leaks from individual channels to controlling information exposure across the whole system.

*The first two authors contributed equally to this work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '23, November 26–30, 2023, Copenhagen, Denmark
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3616655

CCS CONCEPTS

• **Security and privacy → Information flow control**.

KEYWORDS

Side Channels; iOS; Timing; Automated Analysis

ACM Reference Format:
Zihao Wang, Jiale Guan, XiaoFeng Wang, Wenhao Wang, Luyi Xing, and Fares Alharbi. 2023. The Danger of Minimum Exposures: Understanding Cross-App Information Leaks on iOS through Multi-Side-Channel Learning. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3616655

1 INTRODUCTION

On a computing system, the occurrence of any event, such as running an IoT app on a smartphone to unlock the door, will inevitably produce side effects – changes in CPU, memory and network use etc., which could be observed by the party not supposed to know anything about the event, such as a third-party app. These side effects, often including resource access patterns and signals generated by the access, are channels from which protected information can be inferred. Complete removal of such side channels is infeasible for most systems, due to either unbearable cost incurred by additional resource uses for covering access patterns (e.g., a large amount of random traffic to hide a timing channel) or a significant downgrade of the system's functionalities by strict control on data release (e.g., noise added to public statistics, rendering them useless). Addressing this challenge today relies on the common wisdom that the side channel leak can be controlled by suppressing information exposure on each channel (timings, packet size, memory access pattern, power consumption, etc.), to the extent that an attack on the channel becomes less likely to succeed: e.g., limiting the rate of releasing CPU usage, so it more likely reflects the aggregated effect of multiple processes, instead of a specific one [1, 2]. A fundamental problem is that associated with the target event are often not a single channel, but multiple ones across the whole system: running an app leaks information through use patterns of CPU, GPU, memory, network, storage and others. As a result, the side channel leaks of the event should be measured by the joint effect of all these channels, while the existing approaches focusing on individual channels are inadequate at best, as the collective information from the minimum exposure of individual channels could

still be significant, and ineffective at worst, when weakening a side channel might accidentally strengthening another one [46].

Multi-channel inference on iOS. The first attempt that explicitly looks across multiple channels is a side-channel analysis on iOS [53]. Unlike Android, which is known for leaking per process information through its `procfs` [10, 25, 26, 50, 55], iOS is more aggressive in controlling information exposure, only releasing through its APIs global statistics such as CPU, memory, and storage usages. These statistics are summaries of multiple processes' operations and each of them is considered to be too noisy for finding any useful information about an individual process. However, the prior research shows that when extracting features from these channels using symbolic aggregate approximation and bag-of-pattern representation, and collectively analyzing the features using a simple Support Vector Machine, sensitive information such as foreground running apps can still be inferred [53]. These findings have forced Apple to modify the iOS kernel to reduce the rate of releasing such global statistics to mitigate the threat [1, 2].

Still less clear, however, is the adequacy of Apple's current protection – whether there exist other channel combinations from which the same or even more sensitive information can be inferred. Actually, the prior study just provides an example to demonstrate that the collective information from noisy individual channels can also have serious privacy implications. Still we do not know the answers to more fundamental questions: given a computing system, how to determine the relations among different side channels on the system? can we find an optimal subset of these channels to estimate the maximum amount of information exposed from the system? Most importantly, how can we control the side-channel leak from the *whole* system, not just individual channels? Further little has been done so far to understand what inference techniques work most effectively on the heterogeneous channels, which essentially is a learning problem. Solutions to these problems will elevate the current side-channel research, from mitigating leaks from a single channel to protecting privacy of the whole system.

Multi-side-channel learning. In our research, we made a further step to analyze the whole-system side-channel risk, focusing on cross-app information leaks on iOS. At the center of our analysis is an attack called *Mischief* (MultI-Side-CHannel learnIng for sEcret inFerence), which first identifies all iOS APIs potentially exposing sensitive information and then utilizes *Merit Score* [24] to analyze joint information leaks from different combinations of these side channels and an algorithm to search for the optimal subset of the channels disclosing most information. Also we formulate inference on the optimal subset as a Multivariate Time Series (MTS) classification problem, and apply the state-of-the-art MTS classifiers, including Rocket [15] and InceptionTime [21], to solve the problem.

In our study of the iOS 16 SDK, we executed an automated assessment, unearthing 1,053 potential side channels distributed across 83 APIs. Notably, each API represents a structured output, possessing several outputs and consequently, multiple channels, which could potentially disclose sensitive data. Running Mischief on these channels, further we identified a subset of channels that leak out the most information for a given attack purpose, as measured by their joint Merit Score: for example, when the purpose is to identify foreground apps, 7 channels (`wire_count`, `cow_faults`, `en0_ibytes`, `en0_obytes`, `en0_ipackets`, `en0_opackets`, and `user_time`) are in the subset, while for inferring the websites visited by Safari, our approach selects 9 channels (`free_count`, `inactive_count`, `wire_count`, `internal_page_count`, `en0_ibytes`, `en0_obytes`, `en0_ipackets`, `en0_opackets`, and `user_time`). Mischief further learns from these channels sensitive cross-app information, showing that indeed the remedy Apple put in place is fragile. Specifically, we found that an unprivileged malicious app is able to accurately identify foreground running apps, the websites Safari visits and the operations of Apple Home (including opening the door, garage, light and viewing the camera, etc). For foreground app identification, our approach achieves a high accuracy not only in the closed-world scenario (100 known apps), as did in the prior research [53], but also in an open-world setting (500 apps with only 100 fingerprinted), which has never been done before, showing the prowess of the new attack. Our evaluation demonstrates that Mischief is effective (high accuracy), efficient (short execution time), and also robust (e.g., models trained on one device can be used on other devices, as shown in Sec. 4).

Responsible disclosure. In December 2022, we reported the vulnerability to Apple. To aid in the investigation, we provided our code, data, and video demonstrations of our attacks. As of the preparation of this manuscript, the Apple security team is still examining the issue, and we remain in communication with them.

Whole-system leak control. Mitigating side channel leaks today primarily focuses on individual channels (e.g., [48]). Even Apple's kernel patch [1, 2] meant to control the leaks from a set of channels fails to suppress the information exposure from other sets of channels, which allows our attack to succeed. Leveraging our new multi-channel analysis, we developed a novel privacy-protection technique that for the first time, aims at reducing the overall information exposition across the entire system, while minimizing the impacts on its functionalities. Our approach, called *Wheels* (WHolE-systEm Leak Suppression), iteratively identifies an optimal subset of channels across the system to assess its information leak using Merit Score and then reduce the leak from the subset by placing control of a small magnitude (adding a small amount of noise, slowing down data release with a small delay, etc.) on a selected channel with the most impact on the score. This iteration ends when no channel subset with a score higher than a given threshold can be found. The threshold set in our study is meant to degrade the accuracy of an inference to the level of a random guess. For this purpose, we utilized HIVE-COTE [27] to evaluate the effect of an inference under a given Merit Score. HIVE-COTE is known to be a heavyweight yet the most effective solution for MTSC, which has been extensively used as a reference point and an upper bound for measuring the performance of other MTSC methods [15, 16, 21, 31, 40, 45].

Contributions. The contributions of the paper are outlined below:

• *New findings and understandings*. We made the first attempt to systematically analyze the relations across multiple side channels on a computing system and develop an effective way to infer sensitive information from an optimal channel combination. Using our technique, we demonstrate that indeed the state-of-the-art iOS is

still vulnerable to an inference attack, exposing protected information such as foreground apps, websites visited and operations of Apple Home on IoT devices to unauthorized parties. Our findings highlight the importance of side-channel defense on the whole system, not just on individual channels.

• *New techniques.* In addition to the attack/analysis technique, we further developed the first approach to suppressing information leaks on a given target across all channels of a computing system. The effectiveness of the technique was evaluated on iOS. These new techniques are the first step toward more comprehensive control of information leaks on a system, across all its side channels.

2 BACKGROUND

2.1 OS-Level Side Channel Attacks

Unlike the attack that exploits software vulnerabilities to gain direct access to sensitive user data, a side channel attack aims to infer such information by monitoring the observable side effects issued during the execution of a computing system or its applications involving the data. Such side effects may seem harmless, but actually reveal certain artifacts or characteristics of the target information (secret data, user activities, etc.). They come from the sources including caches (cache side channels), sensors (sensor-based side channels), and the APIs open to third-party apps for querying the status of the device, the OS or other apps (OS-level side channels). On a mobile OS like iOS and Android, knowledge about user activities is sensitive and should not be disclosed to unauthorized parties, since the information could be abused for the purposes such as phishing, as reported by the prior studies (e.g., [17]).

The focus of our research is OS-level cross-channel leaks. Prior research has demonstrated that OSes often fail to properly control information exposure from side channels. A prominent example is procfs, a pseudo file system used on UNIX-like OSes (including Android) to export kernel statistics for each process (e.g., memory usage, CPU usage, network usage) to user space. Individual side channels under procfs have long been studied [10, 17, 25, 26, 50, 55]. It does not exist on iOS though, which only releases global statistics to allow third-party apps to monitor resource consumption of the whole system (e.g., CPU, memory, storage usage). These statistics are aggregated data and considered to be "safe", since detailed per process data is hard to recover from each of them. Collectively, however, these public statistics are found to be still quite informative: particularly, user activities are shown to be identifiable from memory, network and file system information, three channels accessible without any permission [53]. In response to the findings, Apple patched the iOS kernel to reduce the frequency of releasing memory usage and cut the precision of network-related statisics [1, 2].

2.2 Multivariate Time Series Classification

Concept. Time series classification (TSC) is a machine learning task in which the input data consists of a series of real-valued, ordered features. For example, stock prices, sales, weather, and web traffic etc. are typical time series data. This problem is challenging since ordered data are known to be hard to analyze and classify. In the past, much attention has been paid to univariate TSC, in which the input is a single time series with corresponding class labels. More useful in the real world, however, is multivariate TSC (MTSC), in which a single input case has multiple dimensions or features, each represented as a time series. Examples of MTSC problems include human activity recognition (predicting user activities from the data collected by wearable devices with a variety of sensors, such as accelerometers, gyroscopes, GPS, electromyography sensors and pressure sensors), diagnosis based on medical data such as ECG or EEG, and monitoring systems. In the context of side channel attacks, MTSC is relevant because an attacker may leverage multiple side channels simultaneously for inferring sensitive information, and each channel can be viewed as a univariate time series (UTS). Following we formally define the multivariate time series (MTS), UTS and the MTSC tasks.

Definition 2.1. A Multivariate Time Series is a sequence of T ordered elements X_i with M dimensions, where each element is a vector of real numbers with M components, i.e. $X_i \in \mathbb{R}^M$. An MTS is represented as $X = [X_1, X_2, \ldots, X_T]$.

Definition 2.2. A Univariate time series X of length T is a type of MTS that has only one dimension, i.e. $M = 1$.

Definition 2.3. The Multivariate Time Series Classification task consists of learning a classifier on a dataset $D = (X^1, Y^1), (X^2, Y^2), \ldots, (X^N, Y^N)$, a collection of pairs (X^i, Y^i), where each pair consists of an input X^i and a corresponding class label Y^i, in order to map from the space of possible inputs X to a probability distribution over the classes Y.

There are two types of MTSC, transformation-based and deep-learning based classification [36], which we elaborate below.

Transformation based methods. Transformation based methods extract features from a training set with transformations and leverage classifiers such as Support Vector Machines (SVMs) or Random Forests to classify the inputs using these features. Such transformations can be dictionary-based or shapelet-based. Dictionary-based transformations for time series classification discriminate time series based on the frequency of repetition of certain sub-sequences. The shapelet transformations classify a time series by identifying and extracting short, repeated sub-sequences (called shapelets) from the series. None of these approaches, however, can achieve state-of-the-art accuracy [36].

Since none of these time series transformation methods (such as shapelets or SFA) can consistently outperform the others, an ensemble of the classifiers operating on different time series representations (called COTE [7]) is considered to be more effective. Particularly, Lines et al. [27] extend COTE with a hierarchical voting scheme, called HIVE-COTE (Hierarchical Vote Collective of Transformation-Based Ensembles), which further improves the performance of the ensemble. HIVE-COTE is currently the most accurate method when evaluated on the UCR archive [13], but its high training complexity (order $O(N^2 \cdot T^4)$) makes it impractical for many real-world applications. By comparison, deep learning models can be trained faster due to their use of GPU parallel computation. So, HIVE-COTE often serves as a reference point or upper bound for evaluating the performance of other time series classification methods [15, 16, 21, 31, 40, 45], as it represents the current best-known accuracy achievable on given datasets.

It is important to note that the prior work on iOS side-channel analysis [53] based only on a single transformation, which could be more efficient than HIVE-COTE but much less effective, in terms of classification accuracy.

Deep learning based methods. Deep learning techniques, which have been successful in tasks such as image recognition and natural language processing, have recently been applied to MTSC. In particular, Convolutional Neural Networks (CNNs) have demonstrated good performance in MTSC. A convolutional layer in a CNN consists of sliding one-dimensional filters over the input time series, allowing the network to extract time-invariant, non-linear features useful for classification. By cascading multiple layers, the network is able to further extract hierarchical features that should in theory improve prediction accuracy.

Currently, the state-of-the-art deep learning approach for MTSC is InceptionTime [21]. InceptionTime is an ensemble of convolutional neural networks that utilize a series of inception modules with residual connections and global average pooling, followed by a fully connected layer. In terms of accuracy, InceptionTime comes close to HIVE-COTE, but it is significantly more efficient, with a training time that is two orders of magnitude faster than HIVE-COTE.

2.3 iOS Cross-App Isolation

System resources access. On iOS, access to protected system resources and data is restricted by default to protect the user's privacy and prevent unauthorized access. All apps from the App Store are sandboxed, which means that they are isolated from the system files, resources, and data of other apps. Developers must request access to these resources and data on a per-case basis, and the user can grant or deny such an access request.

For example, if an app developer wants to access the smart home devices, she needs to include the `NSHomeKitUsageDescription` key in the app's property list file. When the app is invoked first time, a dialog box appears, asking for the permission to access the smart home devices. If the user grants the permission, the app will be able to access the devices. The user can also revoke this permission at any time through the Settings menu. This mechanism allows the user to control which app has access to their data and resources, ensuring that their privacy is respected.

Cross-app communication. There are several ways that iOS apps communicate with each other. One method is through the URL scheme: an app can register a unique URL and by activating it, other apps can pass data to the app. The iOS pasteboard is another channel for cross-app data sharing, which acts as a system-wide clipboard for storing and retrieving data. Since iOS 16, each time an app tries to access the pasteboard, a prompt is displayed to the user asking for permission to do so. This ensures that apps cannot access the pasteboard without the user's consent. For sharing data within the same developer team, apps can utilize App Groups to exchange data such as user defaults, files, and database information. Given that there is no way for sharing data between different developers' apps without the user's consent, monitor activities across these apps cannot be done through these public communication channels.

App vetting. Before an app is available to iOS users, it must go through a review process by Apple to ensure that it meets the App Store Review Guidelines [3]. These guidelines describe the technical, performance, and security requirements that all apps need to follow in order to be accepted to the App Store. As part of this review process, Apple verifies whether apps are using APIs and frameworks for their intended purposes [3]. If an app is found to violate these guidelines, it will not be published.

2.4 Threat Model

In our research, we consider the adversary who aims to infer the events of interest on a target iOS system, such as the websites visited, the operations of IoT devices, and the apps running in the foreground by analyzing the API returns from the iOS kernel.

For this purpose, the adversary is assumed to run an unprivileged attack app on the target system. The app masquerades as a program with a legitimate reason to operate in the background, such as an audio player. Except that, the app does not need any permission that requires explicit user consent.

To mitigate the threat posed by the adversary, the defender intends to control information leaks from all possible side channels related to the target event. At the defender's disposal are a set of standard techniques that can be applied at the OS level to suppress information exposure from the API return, such as adding noise to the return, reducing its accuracy or releasing the data at a lower frequency. The protection is considered to be successful if the attempt to infer information from any subset of the channels on the system is no more effective than a random guess.

As mentioned earlier, our research focuses on OS-level side channels, not the information leaks from micro-architecture, power consumption, electronic magnetic emission, and others, though the same analysis could also be applied to evaluate the collective leaks cross these channels.

3 THE MISCHIEF ATTACK

3.1 Overview

As mentioned earlier, Mischief is designed to utilize an optimal subset of side channels, which collectively expose most information about a target activity on iOS (such as an app running in the foreground), to achieve the most effective identification of the activity. Most challenging here is how to find such an optimal subset. For this purpose, our approach first systematically detects potential OS-level side channels across the system, and then searches for their optimal combination using a correlation analysis. Inference on the combination discovered is formulated as an MTSC problem, which Mischief addresses by building a machine learning (ML) model using the state-of-the-art solution, to accurately predict the occurrence of the event of interest from a time series of observations from all the side channels in the combination. In order to enable the attack in an open world scenario, which has never been done before, Mischief performs data augmentation to better generalize the model. Following we describe these two phases at a high level.

Optimal side-channel combination discovery. Mischief uses a simple approach to find potential iOS side channels from the Apple developer documentation: it goes through the specifications of all iOS APIs to identify those that return statistics (counters) related to a set of keywords; these keywords come from automatic extension of a small set of seeds by a co-occurrence analysis. On

these identified channels, Mischief searches for an optimal subset as measured by Merit Score [24], a metric for correlation-based feature selection. This search is formulated as an optimization problem, which Mischief solves using simulated annealing to approximate the global optimum. This allows Mischief to discover a set of channels that could enable the most powerful inference attack.

Learning for target activity inference. From the channels discovered, Mischief collects a dataset $D = (X^1, Y^1), (X^2, Y^2), \ldots, (X^N, Y^N)$, where Y^i is a class label of an activity of interest (e.g., a specific app running in the foreground) and X^i represents an M-dimensional multivariate time series, with each dimension being a univariate time series of a length T that describes T consecutive observations from a selected channel (e.g., the CPU usages observed in T consecutive time frames). Then our approach utilizes a state-of-the-art deep learning technique to learn an MTSC model for predicting the class label for the activity of interest based on the side channel information captured by the input data X, also an MTS describing T observations from all channels in the optimal combination. The model is expected to effectively distinguish between the inputs X^i characterizing a specific class (e.g., a specific app, a website visited, etc.) and other classes (e.g., other apps, other sites). For this purpose, the model should be well generalized, capable of identifying the class not only from others of interest to the adversary but also from those not being targeted. This requires a lot of training data related to the target activity that may not be practically obtained. To address this challenge, Mischief employs an online data augmentation method that leverages a diverse set of pre-collected background noise samples to improve the model's generality. As a result, Mischief becomes capable of recognizing user activity even in an open world scenario, accurately distinguishing between the inputs of interest and those not.

Fig. 1 illustrates the design of Mischief. The rest of the section elaborates on individual attack phases.

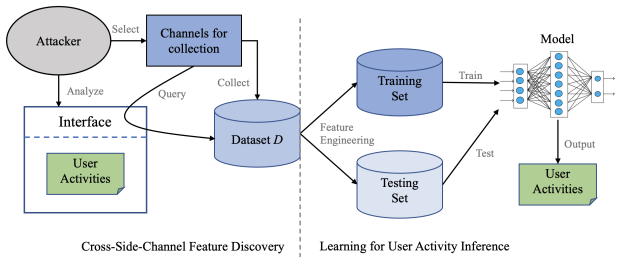

Figure 1: Overview of Mischief. Mischief consists of two main stages: (1) identification and selection of informative side channels, and (2) learning from these channels to infer user activities.

3.2 Optimal Channel Combination Discovery

The first phase of Mischief aims to find the most effective channel combination for inferring the target activity. This phase can be conducted offline, and consists of two stages: systematic channel identification and optimal channel set discovery.

Channel identification. To build an efficient classifier, the first step is to collect a set of iOS APIs that expose information about the target activity. Typically, such information is related to the computing resource (e.g., CPU, memory, disk and network etc.) consumed by the operations related to each class of the each activity. So Mischief takes as its input a list of resource-related keywords, including CPU, memory, storage, and network, and further expand it by adding acronyms and synonyms using Word Net [33]. The expanded resource list is then used to pre-screen APIs.

From the resource-related APIs, we find those returning statistics about the resources related to the target activity. To this end, our approach utilizes an initial keyword vector with a set of keywords such as num, len, count, and time, which can be configured by the user of our technique. This initial vector is then automatically expanded by seeking other keywords highly related to the vector, based upon the number of their co-occurrences in the API specifications with the keywords already in the vector. The expanded vector then serves the purpose of extracting useful statistics from the APIs selected through the pre-screening, by fuzzy-matching the content of API specifications. The statistics discovered in this way are considered to be potential side channels.

In our research, we performed this analysis on the iOS 16 SDK, which contains a large amount of information about all APIs. The resource list and the keyword vector we used are shown in Table 1. Note that each selected API may contain multiple potential side channels, since an API may return a data structure carrying multiple statistics. Each statistic whose value changes with the occurrence of the target activity is considered to be a separate channel.

Table 1: Generated Keywords.

List	Keywords
Resources	container, cpu, device, disk, host, machine, mem, net, network, node, pod, procedure, process, service, storage, thread, virtual memory, vm, volume
Keyword Vector	active, allocated, avail, bit, byte, cnt, count, current, elapsed, free, hit, host, idle, info, len, load, num, offset, packets, receive, request, reserved, resident, sec, send, sent, shared, size, state, statistics, suspend, swap, terminated, tick, time, use, valid, wired

Using identified potential channels, Mischief then constructs the dataset $D = (X^1, Y^1), (X^2, Y^2), \ldots, (X^N, Y^N)$. Such data is gathered by continually querying the selected APIs and labeling their returns with the corresponding activity classes. Based upon this dataset, an optimal channel set is selected for side-channel inference.

Optimal channel set discovery. Given the potential side channels related to a target activity, one may intend to use *all* of them, in the hope of maximizing the amount of information for inferring the activity. This simple approach, however, does not work well, since bundling random channels together may not improve information gain, and instead may even undermine the inference effectiveness, due to the noise introduced by these channels. Further learning from a large number of channels needs more training data and takes more time, adding to the burden of classifier construction. So it is important to carefully select a subset of channels that provide the most valuable information through a channel selection process.

Serving this purpose is *correlation-based feature selection* [24], a method for evaluating the utility of a subset of channels for inferring the target activity, based on *Merit Score*. Intuitively, a set of channels

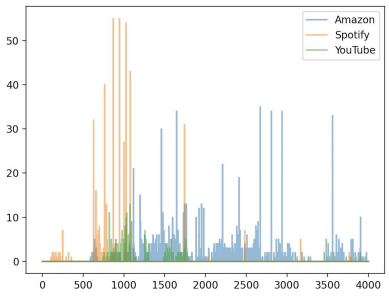

Figure 2: An example of the time series data collected by the en0_opackets channel while three iOS apps were launched.

become most informative when they are strongly correlated with the target activity while less so from each other (so they provide complementary information) [24]. Such a relation characterizing a channel set is measured by Merit Score, as defined below:

$$Merit = \frac{k\overline{Corr_{cf}}}{\sqrt{k + k(k-1)\overline{Corr_{ff}}}}, \tag{1}$$

where $\overline{Corr_{cf}}$ is the average correlation between the features of the information, in terms of time series, observed from the side channels in the subset, and the class label, while $\overline{Corr_{ff}}$ is the average correlation between the selected features; k denotes the number of features in the subset. Note that all the correlations are calculated on the collected dataset D, which is a set of labeled MTS.

For example, Fig. 2 shows the time series data observed from a channel (statistic) `en0_opackets`, the number of packets delivered through the WiFi interface that was gathered by a background app when three iOS apps (Amazon, Spotify, and YouTube) were launched. Each data point in the figure describes the change of the statistic since the last query. To gather the data, the API was called 500 times per second, so the unit of the x-axis is 0.002 seconds. As we can see from the figure, these apps are distinguishable from the time series of the network statistic (though the UTS alone may not work well against a large number of apps), demonstrating the value of using time series to represent information leaks.

To capture the correlation across different time series, we need to extract their features, an important step in time-series analysis [36]. As mentioned earlier, feature extraction using transformations such as shapelets or SFA is computationally expensive, particularly when working on a large dataset. An emerging, more efficient solution is to train a DNN classifier on a time series and use the representation it generates as the feature for the series [15, 16, 21, 45]. In our research, we found that a simplified version of the approach, using only the output layer of the classier, also yields good results (Sec. 4).

Specifically, for a given channel i, our approach first extract from the labeled MTS dataset D a labeled UTS dataset D_i that contains only the time series data of the i-th channel (i.e., the ith dimension of all X in the dataset). Then, we perform K-fold split to train K classifiers for the i-th channel on dataset D_i to predict the class label for each UTS: that is, D_i is randomly broken down into K subsets, with each subset corresponding to a classifier; Classifier κ ($1 \le \kappa \le K$) is used to generate predictions on the κth subset but trained on the remaining $K-1$ subsets. All predicted labels produced by these K classifiers for all univariate time series in D_i form a feature vector for the channel i.

Here is the idea behind our approach: if the time series of the channel i enables classifiers to effectively predict their class labels for the target activity, then this channel must be highly related to the activity, so it contributes to a high average correlation with the target $\overline{Corr_{cf}}$. In the meantime, if another channel j is characterized by the similar prediction results as i, adding j to the combination does not help improve the utility of the combination since it does not provide much new information.

As an example, consider a dataset D with 1000 MTS data points from two channels, CPU and memory usages. Each MTS in the dataset, X^i, is a sequence of 100 elements – the samples collected from the two channels at 100 different time points. So each MTS is a 2×100 matrix. To build the classifiers for each channel, the dataset D is split into two separate datasets, D_{cpu} and D_{mem}, where D_{cpu} contains 1000 CPU usages, each in the form of a 1×100 univariate time series, and D_{mem} involves 1000 memory-usage UTSes. Each UTS in the new datasets retains the activity label of its corresponding MTS in D, so the CPU usage time series can be aligned with the memory usage series if they were all observed when the same activity event happens, such as running the YouTube app in the foreground, visiting Chase bank's website, etc. Each of these two datasets are then broken down into 10 subsets to train 10 different classifiers. The classifiers for the CPU usages then produce the labels for each UTS in D_{cpu}, and so does the classifiers for the memory usages. These labels form a feature vector for the CPU usage and another vector for the memory usage. Their joint contribution to information inference is then estimated by calculating their Merit Score based upon their correlations.

Our current design adopts *Mutual Information* (MI) for calculating the correlation of both $Corr_{cf}$ between a channel and the class label of the target activity and $Corr_{ff}$ between two different channels. MI is an information theoretic concept measuring the dependency between two random variables: i.e., the amount of information about one variable can be derived by observing the other. Given two continuous random variables X and Y, the entropy of X is defined as:

$$H(X) = -\int_x p(x) \log p(x)\, \mathrm{d}x. \tag{2}$$

The joint entropy of X and Y becomes:

$$H(X,Y) = -\int_{x,y} p(x,y) \log p(x,y)\, \mathrm{d}x\, \mathrm{d}y. \tag{3}$$

Then, the MI between X and Y is calculated as:

$$MI(X,Y) = H(X) + H(Y) - H(X,Y). \tag{4}$$

In our research, we used the Adjusted Mutual Information (AMI) [34] score to calculate the correlations. AMI is a variant of mutual information adjusted to account for the possibility of observing the accidental agreement of the two random variables, given the distribution of the data. It is generally more reliable than MI in evaluating the strength of the association between two variables, especially when the sample size is small or the distribution of the data is not well-known. It is defined as:

$$AMI(X,Y) = \frac{MI(X,Y) - E[MI(X,Y)]}{mean(H(X),H(Y)) - E[MI(X,Y)]}. \quad (5)$$

Using Merit Score as a metric, Mischief searches for the optimal combination of potential side channels. Clearly, a brute-force search is exponential and therefore not an option. So we use probabilistic search to approximate the optimal solution. Specifically, our approach is an ε-greedy search, which at any moment, selects a channel and adds it to a subset. The selection is done probabilistically, with a probability ε to choose the channel outside the subset that maximizes the Merit Score with the channels already inside the subset compared with other outside channels, and with a probability $1-\varepsilon$ to choose a random outside channel. This strategy is used in reinforcement learning and we also apply simulated annealing to ε to ensure the convergence of the process and further repeat the whole search process for multiple times to avoid local optima.

Algorithm 1 presents the whole channel selection procedure where Line 1 – 6 build a correlation profile using the classifiers trained on the individual channels and the rest are for the ε-greedy search. The key steps are Line 8 (restarting the whole search procedure for multiple times), Line 14 (randomly picking a channel with probability ε and greedily picking the best channel with a probability $1-\varepsilon$) and Line 25 (performing simulated annealing).

Algorithm 1: Channel Selection for Multi-channel Side Channel Attacks

Input : Multiple side-channel traces X of n channels, labels Y, Restart times T_{max}, Epsilon ε, Annealing rate $\Delta\varepsilon$.
Output : The optimal set of channels $\hat{m}_c$, the best Merit Score $\hat{Merit}$.

1 **for** $0 < i \leq n$ **do**
2 Train classifier f_i on $\{X_i, Y\}$
3 $pred_i \leftarrow f_i(X_i)$
4 $Corr_{ff} \leftarrow AMI(pred_i, pred_i)$
5 $Corr_{cf} \leftarrow AMI(Y, pred_i)$
6 **end**
7 $\hat{m}_c = \{\}$, $iter = 0$, $\hat{Merit} = 0$
8 **while** $iter < T_{max}$ **do**
9 $k \leftarrow 1$, $\Delta Merit \leftarrow 1$
10 **while** $\Delta Merit > 0$ **do**
11 Identify k sized channel subsets $\{c\}$ which includes the selected $k-1$ sized channel subset.
12 Calculate Merit Score $\{Merit\}$ using Equation 1 for each k sized subsets.
13 Sample a $dice$ from 0 to 1.
14 **if** $dice < \varepsilon$ **then**
15 $m_{c\_k}, Merit_k \leftarrow$ random sample a channel to form a k sized subset
16 **else**
17 $m_{c\_k}, Merit_k \leftarrow \arg\max(\{Merit\})$
18 **end**
19 $\Delta Merit \leftarrow Merit_k - Merit_{k-1}$
20 $k \leftarrow k+1$
21 **end**
22 **if** $Merit_{k-2} > \hat{Merit}$ **then**
23 $\hat{m}_c, \hat{Merit} \leftarrow m_{c\_k-2}, Merit_{k-2}$
24 **end**
25 $iter+ = 1$, $\varepsilon- = \Delta\varepsilon$
26 **end**
27 **return** $\hat{m}_c, \hat{Merit}$

3.3 Learning for User Activity Inference

On the selected channels, Mischief builds a model to infer the target activity by modeling it as a Multivariate Time Series Classification problem (as defined in definition 2.3), where the model inputs X^i are the time series data collected by periodically calling the APIs of these selected channels, and the output Y^i is the class label of the target activity. Our classification framework consists of three major components: data pre-processing, data augmentation, and classifier construction, which are elaborated below.

Data pre-processing. To prepare the training data for our MTSC model, our approach first collects samples for each class of the target activity (e.g., a specific app running in the foreground). As mentioned earlier, each sample is a time series for a specific channel, such as CPU usage, memory usage, etc., gathered in the presence of an event (a specific class of the activity). All such samples form the dataset $D = (X^1, Y^1), (X^2, Y^2), \ldots, (X^N, Y^N)$, where X^i is the MTS for all selected channels and Y^i is the class label. To highlight the useful information from the collected time series data, we calculate the difference between consecutive data points for each time series X^k, that is, $diff^i[k] = X^i[k] - X^i[k-1]$. This transformation allows us to focus on the changes in a time series rather than its data points' absolute values.

Data augmentation. A model that can accurately recognize the target events in an open-world scenario should generalize well on its training data, capable of effectively classifying the data outside its training set. For this purpose, a large amount of highly diverse data, including a wide range of examples and variations, is needed to help the model capture the robust and generalized representations of the input space. Such training data, however, can be expensive to generate in practice, particularly for a large number of events related to the user operations (e.g., many apps to be identified). So we applied data augmentation techniques [41] to artificially increase the size and diversity of the training dataset in our research.

However, data augmentation for time series data is often challenging, since a time series describes a sequence of ordered and interdependent observations. So simply adding random noise to the data, would not capture such dependencies and could result in unrealistic or even misleading data [47]. To address this challenge, we designed a unique data augmentation method using "natural noise". Specifically, recall that under our threat model, the adversary relies on global statistics about system resources shared among all apps to infer sensitive user information. So any sample made on such a statistic is affected by both the activity of interest in the foreground and others that take place in the background. To build a model that is able to withstand the noise incurred by these background activities, our approach collects samples of "natural noise" by querying each statistic when running a variety of apps in the presence of a foreground app whose resource consumption is stable, not causing any change to the time series collected: in our research, we utilized the calculator app, which in the absence of interactions, does not incur additional CPU and memory use

and any network traffic. Such noise samples are then applied to the training set for an *online* data augmentation. Online augmentation applies transformation to the input data during the training process, rather than to the training set before the training. This approach allows the model to be trained on more diverse data points, thereby better generalizing the model.

Classifier. As mentioned earlier, we formulate the multi-channel-based side channel attack as a Multivariate Time Series Classification problem, to which InceptionTime [21] is a state-of-the-art solution (Sec. 2.2). InceptionTime achieves a high classification accuracy, coming even close to the much more heavyweight HIVE-COTE, using a combination of Inception modules and ResNet, and by combining the predictions of multiple models of this architecture trained with different random initial weights. The InceptionTime architecture include two blocks, each with three Inception modules [44], together with residual connections and global average pooling and softmax layers. This architecture enhances the stability and performance of a model.

Another MTSC solution is Rocket [15], which is built upon a large number of random convolution kernels together with a linear classifier (such as ridge regression or logistic regression). The kernels are applied to individual instances, and from the resulting feature maps, the maximum value and a new feature – the Proportion of Positive Values (PPV), are returned. Rocket is also a state-of-the-art technique, which performs competitively with InceptionTime. We chose to use InceptionTime over Rocket because InceptionTime is more suitable for online data augmentation. Specifically, training of deep learning models such as InceptionTime requires a large number of iterations, so under online augmentation, the model is exposed to a large amount of diverse training inputs, as each iteration comes with an input transformation: i.e., adding different natural noise samples to the input time series. By comparison, transformation-based methods like Rocket do not go through iteration rounds, so they can only use offline data augmentation, which renders these models less generalized on a small amount of training data and therefore are less ready to work in an open-world scenario. We compared these approaches in our research (Sec. 4).

4 EVALUATION AND FINDINGS

4.1 Experimental Setup

Hardware and software settings. All our experiments were conducted on iPhone XR and iPhone 11 that were not jailbroken and were running iOS 14.2 and iOS 16.2 [1] respectively. Our monitoring app was set to call APIs at a frequency of 500 times per second, while each activity class (e.g., running a specific app) was monitored for a duration of 10 seconds, which produced a time series with 5,000 data points. Note that the data were always collected immediately after the launching time of each app and therefore only the first 5,000 data points after app launching are monitored.

For data augmentation, we collected the natural noise samples by running the iOS-provided Calculator app in the foreground and the top 100 most reviewed free apps in the background. We made two samples across all side channels for each background app. This results in 200 noise samples for transformations in online data augmentation, in which the noise was added to the input to the model under training.

Mischief configuration. For the channel selection stage (Algorithm 1), we set the parameters as follows: $\varepsilon = 0.2$, $\Delta\varepsilon = 0.02$, $T_{max} = 10$ in all our experiments. In the training stage, we constructed an InceptionTime model with two Inception blocks, setting the default values for the model as `bottleneck_size=32, filter_length={5,11,23}, filter_num={32}`. The model was trained on the pre-collected dataset for 300 epochs with a batch size 16, the initial learning rate of its Adam optimizer being 1×10^{-3} and weight decay being 1×10^{-5}. For Rocket, we used 10,000 random convolutional kernels to transform data, and ridge regression to build the classifier.

App store vetting. In our research, we submitted a monitoring app capable of conducting all 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. Our app successfully passed the vetting, which indicates that the program capable of launching the Mischief attack is not considered to be malicious by Apple. It is worth noting that we immediately withdrew our application from the App Store right after it was approved. According to our dashboard, no downloads occurred and no users were affected.

4.2 Effectiveness and Discoveries

Our attack focuses on two types of activities: apps running in the foreground and in-app operations. Note that these activities are related: finding in-app operations is contingent upon recognition of the foreground app. However, for simplicity of analysis, these two activities were studied and related attack techniques were evaluated separately in our research. In this paper, for in-app operations, we focus on two specific activities: visiting websites and communicating with IoT devices.

Foreground app inference. To determine the apps running in the foreground, our implementation of Mischief focuses on the top 100 most reviewed free apps from the App Store, though any iOS apps, including those sensitive ones (LGBTQ+ dating app, porn app, etc.), can be targeted using our technique. Also note that the adversary can first identify the apps with sensitive functionalities, such as healthcare, payment, web-surfing, etc., and then infer when related operations have been executed, for example, visiting a website for a specific disease. We utilized those popular apps in our research since they involve a wide range of functionalities and can therefore serve as a benchmark to evaluate the effectiveness of our attack technique. Data was collected for each targeted foreground app, with each app being launched 50 times while the monitoring app ran in the background. The collected time series data was then labeled with the app running in the foreground. The final dataset contains 5,000 samples. The data was split into training and test sets with a ratio of 8:2, i.e., the dataset includes 4,000 training samples and 1,000 test samples. Table 2 demonstrates that our attack can achieve a 94.1% classification accuracy on the test set, indicating their ability to accurately distinguish between various foreground apps. This poses a significant security risk.

[1] iOS 16.2 was released on December 13, 2022, which is the most current version as of the date this paper was written.

Table 2: Optimal channel set for different events.

Events	Channels	Acc. (%)
Foreground Apps	en0_ibytes, en0_obytes, en0_ipackets, en0_opackets, user_time, wire_count, cow_faults	94.1
Websites	en0_ibytes, en0_obytes, en0_ipackets, en0_opackets, user_time, wire_count, free_count, internal_page_count, inactive_count	93.7
IoT activities	en0_ibytes, en0_obytes, en0_ipackets, en0_opackets, user_time, wire_count, faults, internal_page_count	91.0

Website identification. To evaluate the inference of the websites visited, we selected the top 100 websites from Moz [6] and visited them using Safari, including Google, Wikipedia, PayPal, etc. Data was collected for each targeted website, with each website being visited 50 times while the monitoring app ran in the background. The collected time series data was then labeled with the corresponding website. The dataset includes a total of 5,000 samples, with 4,000 allocated for training and 1,000 for testing. The result (Table 2) shows that our attack achieves an accuracy of 93.7% in classifying websites, indicating a significant security concern. For example, visiting PayPal could indicate a payment transaction, allowing an adversary to identify the user by matching the payment time with public records.

IoT operation detection. To analyze the disclosure of the operations on IoT devices, we used 10 popular devices that can be controlled through Apple HomeKit. Specifically, we conducted experiments on 6 IoT devices including a smart lock, a surveillance camera, a LED light, a thermostat, a garage door opener, and a smart doorbell connected to Apple Home, in an attempt to identify 10 different types of commands issued by Apple Home to these devices, including opening the door, viewing the surveillance camera image, checking the motion detector of the surveillance camera, checking the light detector of the surveillance camera, turning on the LED light, checking the thermostat, opening the garage door, viewing the smart doorbell image, checking the motion detector of the smart doorbell, and checking the light detector of the smart doorbell. Data was collected for each command issued by Apple Home, with each command being triggered 50 times while the monitoring app ran in the background. The collected time series data was then labeled with the corresponding command. The dataset has a total of 500 samples, with 400 designated for training and 100 for testing. Our experiment result (Table 2) shows that our attack achieves an accuracy of 91.0% in classifying different commands issued by Apple Home. This is concerning as it allows the adversary to infer highly sensitive information about the IoT user, such as when her home door or garage door is open.

Selected channels. As shown in Table 2, the optimal set we discovered includes the statistics for network, memory and CPU. Specifically, statistics `en0_ibytes`, `en0_obytes`, `en0_ipackets`, and `en0_opackets` are network side channels exposed through the API `getifaddrs()`. `en0_ibytes`, and `en0_ipackets` keep track of the number of bytes/packets received by the device over the network, and `en0_obytes` and `en0_opackets` count the number of bytes/packets transmitted by the device to the network. These four channels are often used together to measure network traffic and can be leveraged to identify patterns and anomalies in network usage. These channels provide complementary information and can therefore be used together for inferring target events.

Memory channels include `cow_faults`, `internal_page_count`, `inactive_count`, `faults`, `free_count`, and `wire_count`. These counters are returned by the API `host_statistics64()`. Such channels provide different amount of information for different events (that is, different classes of the target activity, such as specific foreground apps to be identified), as measured by their correlations with the events. This could be attributed to the characteristics of these events and how they affect the usages of memory and resources on the device. In particular, the `cow_faults` counter, which records the number of copy-on-write faults, is shown to be most effective in identifying foreground apps, since foreground apps tend to have diverse behavior and therefore incur a broad variety of memory operations, as reflected by the COW faults caused by these operations.

Also the `faults` channel monitors the number of page faults that have occurred, which turns out to be very useful for inferring IoT events, as such events (e.g., camera streaming) tend to have distinct patterns in memory and disk I/O accesses, leaving their footprints in the number of page faults incurred. Although Safari utilizes page caching to quickly retrieve recently visited pages, thereby reducing the number of page faults, page caching causes a large number of memory pages to be kept in physical memory, affecting `inactive_count` and `free_count`. These two counters keep track of the number of pages that have been recently used but can be quickly reused if needed and the number of pages that are not being used and do not contain any useful data, respectively. They are more effective, compared with `faults` for identifying visited websites.

The `internal_page_count` channel records the number of pages that have been loaded into memory. A foreground app tends to use a lot of memory pages, causing `internal_page_count` to be easily saturated and thereby losing its identification power. When it comes to IoT events and website visits, however, the count becomes more volatile. This could result in the `internal_page_count` channel to be included in the optimal set for inferring IoT and website activities, not in the set for foreground app detection. The `wire_count` channel tracks the number of pages that are "wired down": that is, they cannot be moved or paged out of memory. This statistic reveals the amount of memory used by system-level processes. It can also serve as a useful indicator for the memory used by GPU.

The CPU channel that is selected is `user_time`, specifically, `cpu_load_info.user_time.microseconds`, which indicates the CPU usage and is returned by the API `thread_info()`. As different activities have different patterns of CPU usage, the channel provides useful information for all events of interest.

Comparative study. In our research, we evaluated the effectiveness of multi-channel inference on foreground app identification, by comparing the approach with the inference performed on individual channels and on a sub-optimal channel set proposed by the prior work [53]. The result shows that a single channel does not provide enough information to accurately identify foreground apps (Fig. 3), with the highest accuracy attained being 73.7%. This

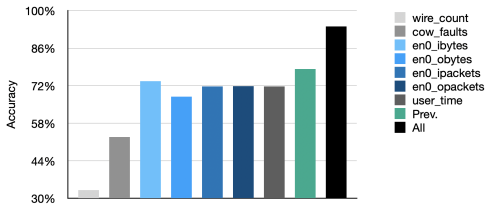

Figure 3: Foreground app recognition results using different channel combinations.

value also corresponds to the performance of the global best single channel.

When all 7 channels are used together, our MTSC achieves an accuracy of 94.1% (shown as the "All" bar in Fig. 3). Also Mischief significantly outperforms the inference attack on the channel set selected by the prior study [53], which includes `en0_ibytes`, `en0_obytes` from the API `getifaddrs()`, and `free_count`, `active_count`, `zero_fill_count`, `faults` from the API `host_statistics64()` (shown as the "Prev." bar in Fig. 3). Note that only two of these channels (`en0_ibytes`, `en0_obytes`) are also selected by our approach. As shown in the figure, such a channel combination only achieves an accuracy of 78.2%, marginally improving over the performance of the single channel inference even in the close world, which indicates that the protection Apple puts in place might work against this known attack. However, the success of the Mischief attack gives strong evidence that such protection is far from enough for controlling side-channel leaks on iOS.

Further, we compared the effectiveness of different learning techniques, including InceptionTime, Rocket and the transformation-based approach proposed by the prior research [53], on the optimal set of channels selected. The prior approach converts time series data into Symbolic Aggregate approXimation (SAX) strings to create a Bag-of-Patterns (BoP) representation, and then trains a SVM classifier on the transformed data. This method is referred to as "SAX+BOP+SVM". Table 3 presents the results of this comparative study, showing that state-of-the-art MTSC methods significantly outperform the prior approach, which again indicates that Apple's side-channel remedy based upon the prior attack is inadequate. Also, while some literature suggests that Rocket beats InceptionTime in some MTSC tasks, for foreground app identification, InceptionTime is found to be more effective, achieving an accuracy (in the close world) 5.4% above that of Rocket. The results also show that online data augmentation (referenced as "InceptionTime + Aug." in Table 3) further elevates the accuracy of InceptionTime by 2.6%.

Open-world results. Prior research all assumes a "closed-world" scenario, in which the adversary has knowledge about all classes of the target activity, such as all the apps deployed on the user's iPhone [53]. In our research, we studied Mischief under a more realistic yet much more challenging *open-world* setting [37], where the adversary only trains a model on a subspace of the target activity, while the rest of space is completely unknown: e.g., only a subset of the apps on the victim's phone analyzed by the adversary before.

To evaluate our attack in an open-world setting, we built an additional *test* set with the time-series traces collected from additional 500 apps. These apps are ranked between 100 and 600 in terms of the number of reviews they have received. To gather their traces, we made two samples from the APIs when each of these apps ran in the foreground, using the same sample rate for analyzing the top 100 apps (500 times per second by default). In total, we gathered 1,000 samples with high diversity (generated by 500 different apps), which are expected to be classified as "unknown" by the adversary, as they are not of interest to the adversary. Specifically, the MTSC deployed by Mischief is expected to assign a given time series to 101 classes: 100 foreground apps and one "unknown" class. It is important to note that the classifier has only been trained on samples from the first 100 classes, not those in the unknown class.

The main idea here is that the classifier should have a higher confidence level for the samples it has been trained on (closed-world apps) compared with those it has never seen before (open-world apps). Therefore, we use the classifier's uncertainty to identify input data that belongs to the "unknown" class. We calculate the classifier's uncertainty by measuring Shannon entropy [39] on the classifier's output.

Table 3: Performance of different methods.

Methods	Closed-world Acc. (%)	Open-world Acc. (%)	Open-world Precision (%)	Open-world Recall (%)	AUC
SAX+BOP+SVM	55.8	61.6	66.9	33.7	0.622
Rocket	86.1	51.0	17.3	5.9	0.557
InceptionTime	91.5	77.4	98.8	73.4	0.760
InceptionTime+Aug.	94.1	86.0	98.9	84.7	0.873

Using the confidence level estimation, we constructed an MTSC and evaluated its effectiveness in distinguishing the samples from the 101 classes. The results are presented in Table 3. As we can see from the table, in the open world, the performance of both Rocket and InceptionTime degrade significantly with SAX+BOP+SVM improving marginally. In the meantime, the online augmentation is demonstrated to boost the accuracy of InceptionTime to 86%. Note that the test set samples may not be evenly distributed among classes, and therefore the accuracy measured can be overestimated: e.g., simply labeling all inputs as "unknown" would achieve an accuracy of 50%. To address this issue, we further measured the outcomes of the classifiers using precision and recall across all 101 classes, as shown in Table 3. As we can see from the table, InceptionTime achieves a high precision on classification (nearly 99%) and augmentation further improves its recall (from 73.4% to 84.7%).

We further group all the user activities of interest (closed-world apps) into a meta-class called "known apps" and use the area under curve (AUC) to measure the open-world attack (Table 3). The result shows that InceptionTime+Aug achieves a high AUC of 0.873, which provides further evidence that the open-world attack is practical.

Transferability. We studied the transferability of our attack across different devices and iOS versions by building our model on an iPhone XR running iOS 14.2 and evaluating it on an iPhone 11 running iOS 16.2. In the experiment, we randomly selected 50 apps from the top 100 apps, launched each app 5 times on iPhone 11 with a monitoring app running in the background, and collected 250 traces for testing. On these traces, Mischief achieves an accuracy

of 88.0%. This mild performance degrade could be attributed to updates of the apps after our collection of training data and before gathering of the test data, in addition to the change of the device and OS version. The transferability of the MTSC could be improved by training it on the data from different devices.

Table 4: Impact of query frequency on attack accuracy.

Frequency	Channels	Accuracy
500Hz	en0_ibytes, en0_obytes, en0_ipackets, en0_opackets, user_time, wire_count, cow_faults	91.5
200Hz	en0_ibytes, en0_obytes, en0_ipackets, en0_opackets, user_time, wire_count, cow_faults	91.3
100Hz	en0_ibytes, en0_obytes, en0_ipackets, en0_opackets, user_time, wire_count, cow_faults	89.2
50Hz	en0_ibytes, en0_obytes, en0_ipackets, en0_opackets, wire_count, cow_faults	88.1
20Hz	en0_ibytes, en0_obytes, en0_ipackets, en0_opackets, wire_count, cow_faults, zero_fill_count	87.6
10Hz	en0_ibytes, en0_ipackets, en0_opackets, wire_count, cow_faults, zero_fill_count	83.0
5Hz	en0_ibytes, en0_ipackets, en0_opackets, wire_count, cow_faults, zero_fill_count	78.0

Effect of query frequency. In our attack's default configuration, the APIs are invoked at a rate of 500 Hz. To examine the impact of query frequency on performance, we executed the attack under various frequency settings without data augmentation. As presented in Table 4, the optimal channel set changes in response to different query frequencies. When the frequency decreases to 50 Hz, the CPU channel `user_time` is eliminated, indicating that the CPU channel is most susceptible to the frequency change among the seven chosen channels. With the frequency going down to 20 Hz, the memory channel `zero_fill_count` is selected, indicating that compared with other channels, this one is less affected by frequency variations. As the frequency further declines to 10 Hz, the network channel `en0_obytes` becomes ineffective and is removed from the optimal set. Interestingly, during this process, with the query frequency dropped from 500 Hz to 5 Hz, we observe that the attack just becomes slightly less effective (accuracy from 91.5% to 78.0%), which demonstrates that the information leak across multiple channels cannot be easily controlled by solely reducing the query frequency, as implemented by Apple to address the prior attack [53].

Power Consumption. We assessed the power consumption of our monitoring app while gathering side-channel data on an iPhone XR running iOS 14.2. Throughout the experiment, the monitoring app was executed in the foreground, calling APIs with different frequencies. We then compared the device when running the app against the situations when running each of four benchmark apps: Spotify, YouTube, Amazon, and when the phone is idle (that is, when the display is active and no app is running), in terms of power consumption. Power consumption was measured utilizing a multimeter tester. The outcomes are illustrated in Fig. 4. Our findings indicate that the monitoring app's power consumption is consistently lower than that caused by each of the benchmark apps, with a maximum of 1.456 W and a minimum of 1.276 W.

Execution Time. We measured the execution time of our attack using the InceptionTime classifier, on a dataset of foreground apps with 4,000 training samples and 1,000 testing samples. Training the

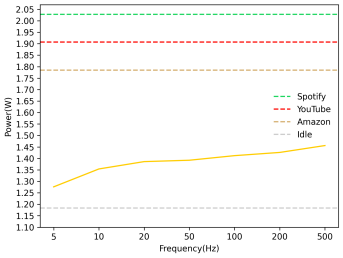

Figure 4: Power consumption of monitoring app under different query frequencies.

InceptionTime classifier for 300 epochs took 33 minutes and 9.9 seconds. Inferring the 1,000 testing samples took 1.89 seconds.

Use of the side-channel APIs in real-world apps. We performed a static analysis on 5,857 iOS apps using Radare [4]. These apps were downloaded between Nov. 17, 2022 and Jan. 6, 2023. Among them, we found `host_statistics64()` in 5,091 of these apps, `getifaddrs()` in 4,940 and `thread_info()` in 4,754. Particularly, 4,243 apps include all 3 APIs which are sufficient for aforementioned attack. The findings show that these side-channel APIs are quite pervasive in legitimate apps, making it hard to detect the malicious app performing the Mischief attack.

Effect of dataset size. We further examined the effect of the size of the attacker's dataset. The experiments were conducted in the closed-world scenario, where the goal was to identify the app running in the foreground, using a dataset of the top 100 popular apps as described in Sec. 4. Our attack's default configuration involves collecting 5,000 samples for foreground apps, with 4,000 samples designated as the training set and 1,000 samples used as the test set. To investigate the effect of dataset size on performance, we kept the test set size constant and gradually reduced the training set size to observe changes in performance. The results of this analysis are presented in Fig. 5.

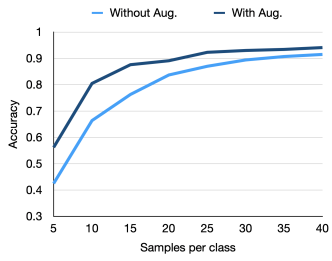

Figure 5: Impact of the number of samples on attack accuracy.

As shown in Fig. 5, reducing the dataset size to 25 samples per class (or 2,500 samples total) has minimal impact on test accuracy. However, when the training set falls below 15 samples per class, the accuracy drops significantly. It's worth noting that previous work [53] has only used 8 samples per class, which greatly underestimates the potential side channel leakage. Additionally, applying

data augmentation techniques can significantly improve the efficiency of the attack. Despite using only 200 samples for data augmentation, it greatly improves accuracy, particularly when the dataset is small. This allows for a quick and lightweight attack, making it more flexible for the attacker.

5 MITIGATION

Existing prevention-based mitigation techniques, such as those adopted by Apple [1, 2] and the differential privacy (DP)-based method [48], typically aim at mitigating the information leak from a single side channel, and are not effective in controlling the inference risk of the whole system, across multiple side channels. To address the problem, we designed and implemented a mitigation technique called *Wheels* (WHolE-systEm Leak Suppression), as a whole-system side-channel control solution. We describe the technique in this section.

5.1 Wheels: Design and Implementation

The goal of Wheels is to determine the set of channels requiring side-channel control and the appropriate level of protection to be applied, so as to suppress the information exposure of a given system with the minimum disruption to the functionalities of related APIs. For this purpose, Wheels employs a Progressive Defense Search (PDS), which iteratively identifies the optimal subset of channels in the system in terms of information exposure, evaluates their joint side-channel leaks using Merit Score, and then reduces the leaks by applying control of a small magnitude on the channel contributing most to the score. Such control can be achieved by reducing the data releasing rate, lowering the granularity of the released statistics or adding noise to the original data. Our current implementation utilizes noise-adding for the protection, as elaborated below.

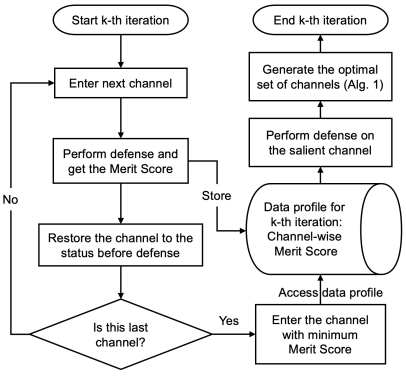

Figure 6: Flowchart of the k-th iteration of PDS ($n_c = 1$).

Progressive defense search. Specifically, given a set of potential side channels on a computing system, PDS iteratively identifies the optimal subset of the channels using Algorithm 1, then finds the n_c most salient channels from the subset ($n_c = 1$ by default), and further adds noise to the channel(s) to lower down the Merit Score of the subset, in an attempt to reduce the highest Merit Score achievable across all the channels on the system, until the score reaches a pre-defined threshold. Fig. 6 presents the flowchart of the k-th iteration of PDS. After the optimal channel subset is found, our approach tests each channel in the subset to understand its impact on the Merit Score, by first adding a small amount of noise to it to calculate the new score of the subset and then removing the noise from the channel and recording it if it is among the top n_c most impactful channels discovered so far. After all channels in the subset have been tested, noise (or other defense) has been applied to the top n_c channels. Then our approach re-determines the new optimal channel subset and if the score of the subset is still above the threshold, the search moves into the next iteration.

Our implementation adds spike noise [47] to a channel to suppress its side-channel leak, with both the direction (addition or subtraction) of the noise and its magnitude being randomly selected. Specifically, for a time series $X = [X_1, X_2, \ldots, X_T]$, as specified in Definition 2.2, the noise generated is added to each element X_i: $\tilde{X_i} = X_i + r_i$, where r_i is randomly sampled according to the Laplace distribution: $r_i \sim Lap(x|\mu = 0, \lambda)$, where λ is the scale parameter and μ is the mean. The level of defense is then determined by the scale parameter λ. We incrementally add a fixed $\Delta\lambda$ determined for each channel, which is proportional to $\max(X) - \min(X)$.

Algorithm 2: Progressive defense search.

Input : Multiple side-channel traces X of n channels, labels Y, target Merit Score T_m, defense step $\Delta\lambda$.
Output: The optimal channel-wise defense $\hat{m}_\lambda$.

1 $\hat{m}_\lambda = \{\}$
2 $\hat{m}_c, \hat{Merit} \leftarrow channel_{selection}(X, Y)$ (Algorithm 1)
3 **while** $\hat{Merit} > T_m$ **do**
4 Calculate Merit Score $\{Merit\}$ after adding Δd defense to each channel of $\hat{m}_c$.
5 Identify the salient channel $k \leftarrow argmin(\{Merit\})$
6 $\hat{m}_\lambda[k] \leftarrow \hat{m}_\lambda[k] + \Delta\lambda$
7 $\hat{m}_c, \hat{Merit} \leftarrow channel_{selection}(X, Y)$ (Algorithm 1)
8 **end**
9 **return** $\hat{m}_\lambda$

Algorithm 2 describes the whole progressive defense search procedure. For each iteration, the noise Δd is added to an identified channel at Line 6. The identification of the salient channel is conducted upon the optimal subset of channels discovered by Algorithm 1 at Line 7. A target Merit Score T_m is the threshold for determining when to terminate the algorithm, which indicates the maximum information leaks tolerable across the whole system. As mentioned earlier (Sec. 2.2), the threshold is estimated using HIVE-COTE, an MTSC known to be heavyweight but provide the most accurate solution to the MTS classification problem. Again, we consider the side-channel leaks to be fully controlled when HIVE-COTE on the optimal channel subset can only attain an accuracy level of random guess. The Merit Score of the optimal subset is then treated as T_m. In our research, we used the dataset of foreground apps (as discussed in Sec. 4) with 100 classes, thus the expected inference accuracy after defense should be around 0.01%. This dataset is representative and has a good transferability across different devices (as described in Sec. 4).

It is important to note that in each iteration, we generate a new attack profile based on the current defense, which is then utilized

to generate the subsequent defense. Consequently, our defense remains effective even if the attacker is fully aware of our strategy.

Furthermore, regarding OS-level channels, we implement the defense by adding numerical noise to the API outputs. In contrast, for other scenarios such as micro-architectural side channels, the defense should be incorporated through alternative atomic operations, such as introducing fake keystrokes [38] or injecting random cache accesses [9]. In these cases, Wheels can evaluate the sufficiency of existing protection and determine the optimal quantity of injected atomic operations, such as fake keystrokes or random cache accesses.

Cross-API consistency. A risk of adding noise to API returns is that the protection may cause inconsistency across the returns of different APIs, which could affect the operations of the app relying on such cross-API invariants. The problem has been studied in the prior research [48], which applies differential privacy to procfs. In our research, we made a preliminary step to understanding the problem and mitigating this utility risk, using the solution provided by the prior research [48].

Specifically, Wheels generates invariants using the foreground app dataset (Sec. 4), which includes the time series traces of the top 100 apps gathered from selected APIs. These traces are analyzed using Daikon [20] to extract invariants. For this purpose, Daikon was configured in our research with a set of templates, referred to as filters. To find single-channel invariants, filters were set to identify the channels that increase monotonically, or decrease monotonically. For multi-channel invariants, we used a filter for detecting linear invariants across all the channels (memory, network, CPU, and storage) in the subset. After Daikon extracted the invariants, we manually examined the outputs, discarding those already implied by other invariants or considered spurious. Given the invariants discovered, Wheels uses them as constraints and runs an integer programming to adjust the randomly generated noise so as to ensure that the invariants are maintained within a channel and across multiple channels in the APIs.

This approach is still preliminary, which may lead to false positives (fake invariants) and false negatives (missing invariants). Further research is expected to understand the impact of the inconsistency risk and more effectively mitigate the risk.

Table 5: Results of Wheels on protecting different events.

Events	Acc. before Defense (%)	Random Guess Acc. (%)	Acc. after defense (%)
Foreground Apps	94.1	1.0	1.9
Websites	93.7	1.0	1.6
IoT activities	91.0	10.0	11.0

5.2 Experimental Results

In our research, we evaluated Wheels from two perspectives: security and utility. For security, we studied the capability of Wheels to defend against side-channel attacks as outlined in Sec. 4. To measure utility, we calculated the relative error between the outputs generated by Wheels and the original output. We performed our experiments on the foreground app dataset as described in Sec. 4. The target merit score, T_m, was set to 5×10^{-3} in the experiments.

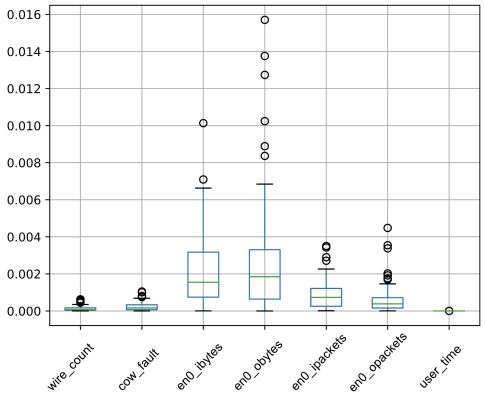

Figure 7: Relative errors. The box in each box plot represents the first, second (median), and third quartiles, with the horizontal lines indicating the range of these values. The whiskers extend to cover all data points that fall within 1.5 times the interquartile range. Outliers are represented by circle ("o") symbols.

We executed the Mischief attack on the APIs protected by Wheels with noised outputs in the closed-world setting, as described in Sec. 4. The results are presented in Table 5. As shown in the table, all the attacks are reduced to random guess, which demonstrates the effectiveness of the protection.

We also measured the impact of Wheels on the utility of the protected APIs. Specifically, we calculated the relative error introduced by Wheels with regards to the original returns of the APIs. Fig. 7 shows the relative errors of the selected channels in Table 2 using 100 data points. As we can see from the figure, the majority of errors introduced by Wheels are small, with 75% of them being less than 0.4%, though some outliers go up to 1.6%.

6 RELATED WORK

Several studies have investigated the potential for side-channel attacks on Android devices. For example, it is possible to extract information about running processes by accessing `procfs` and `sysfs` [10, 18, 25, 26, 43, 50, 52, 55]. However, these studies have only examined single-channel attacks and has not addressed how to effectively combine multiple channels, which leads to an underestimation of the threat posed by multi-side-channel attacks. The research in [53] is the initial examination of multi-side-channel attacks on iOS and shares similarities with our study. However, their approach of manually selecting channels and directly putting them together does not provide a comprehensive understanding of the multi-side-channel threat. As demonstrated in Sec. 4, their approach is unlikely to be successful in current versions of iOS.

Additionally, aside from side-channel attacks on iOS, several studies concentrate on combining multiple side channels [14, 19, 22, 30]. However, akin to [53], these studies also focus on manually selected channels and employ methods specifically designed for those channels. In contrast, we propose a more general approach that automatically selects optimal channel combinations likely to leak the most information, using more generic techniques (deep

learning) to achieve this goal. We believe that this concept can effectively address profiling attacks on other platforms as well. We reserve this exploration for future work.

Prior to our work, several studies have concentrated on using mobile sensors to infer personal information of users. For instance, Zhou et al. [54] and Yu et al. [49] utilized microphone to eavesdrop the lock pattern and keystrokes. Chen et al. [11] and Michalevsky et al. [32] used the battery consumption data to infer the apps and device location. Zhang et al. [51], Ba et al. [6], and Anand et al. [5] leveraged accelerometer and gyroscope to exploit the side-channel information of the smartphone. However, iOS provides security features that make it more difficult for apps to perform these attacks without the user's knowledge. Since iOS 14, an orange icon will appear on the screen whenever any app is using the microphone, making it easier for users to identify and stop any suspicious activity on their device. Additionally, iOS does not provide detailed battery usage data to production apps. To access the accelerometer, magnetometer, or gyroscope, apps must request permission from the user, and the user will be prompted to grant or deny access when the app first attempts to access the device's motion data. Therefore, any attack utilizing these sensors would only be feasible if the attacker's app has obtained access through user permission, making it less stealthy and less practical on iOS.

Numerous studies show that website and/or app fingerprinting can be performed by exploiting micro-architectural side channels, such as cache timings [35, 42], and interrupt timings [12, 28]. However, these attacks have only been executed on desktop platforms, and it remains uncertain whether they can be applied to iOS devices, which would require extensive reverse-engineering of the Apple device's hardware configuration. Side channel attacks on iOS devices are challenging for two reasons. First, Apple conceals the underlying complexities of their devices from both users and app developers, and does not provide comprehensive documentation about the system. Therefore, security research often starts with reverse engineering. Secondly, dynamic analysis is hard on iOS devices, since it requires compromising the operating system and removing security restrictions to debug arbitrary applications [8]. Even then, kernel debugging is often not possible. These limitations make understanding hardware configurations challenging. Furthermore, microarchitectural side channel attacks need to be tailored to the specific hardware configuration of the targeted system [42], whereas our attack is meant to be hardware-agnostic.

Further, besides the OS-level attacks, there is little success reported on iOS by prior research, with only two exceptions focusing on electromagnetic side-channel attacks [23, 29]. Note that these two studies are based upon different threat models than ours. Specifically, Genkin et al. [23] reveals that by employing magnetic probes in close proximity to the iPhone or power probes connected to its USB cable, ECDSA keys utilized in OpenSSL and CoreBitcoin on iPhones can be extracted. Lisovets et al. [29] shows that Apple's proprietary AES coprocessor hardware is also susceptible to electromagnetic side-channel attacks. However, the attack reported by Lisovets et al. [29] was performed on old iPhone models (e.g., iPhone 4) and is deemed infeasible for iPhone 6 and newer versions. Furthermore, both studies require the attacker to have physical access to the device, which is much stronger than our threat model.

7 CONCLUSIONS

In this paper, we introduced the Mischief attack, which systematically analyzes multi-side-channel leakages. We effectively conducted real world attacks on today's iOS, including identifying foreground apps, website visits, and sensitive IoT operations. These examples show that the information leakage from multiple channels poses a real risk to users' privacy. To address the issue, we proposed Wheels, an approach for multi-side-channel leakage control which is applied to the entire system. The experiments show that Wheels can defend against the proposed attacks while preserving the original functionality of iOS APIs. Our research provides guidance for future defenses against multi-channel combination attacks.

ACKNOWLEDGEMENTS

This paper is supported in part by NSF CNS-196030.

REFERENCES

[1] 2017. CVE-2017-13852. Available from MITRE, CVE-ID CVE-2017-13852. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-13852
[2] 2017. CVE-2017-13873. Available from MITRE, CVE-ID CVE-2017-13873. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-13873
[3] 2022. *App Store Review Guidelines.* Retrieved Dec 30, 2022 from https://developer.apple.com/app-store/review/guidelines/
[4] 2022. *Radare.* Retrieved Dec 30, 2022 from https://www.radare.org/r/
[5] S Abhishek Anand, Chen Wang, Jian Liu, Nitesh Saxena, and Yingying Chen. 2021. Spearphone: a lightweight speech privacy exploit via accelerometer-sensed reverberations from smartphone loudspeakers. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks.* 288–299.
[6] Zhongjie Ba, Tianhang Zheng, Xinyu Zhang, Zhan Qin, Baochun Li, Xue Liu, and Kui Ren. 2020. Learning-based Practical Smartphone Eavesdropping with Built-in Accelerometer.. In *NDSS.*
[7] Anthony J. Bagnall, Jason Lines, Jon Hills, and Aaron Bostrom. 2015. Time-Series Classification with COTE: The Collective of Transformation-Based Ensembles. *IEEE Trans. Knowl. Data Eng.* 27, 9 (2015), 2522–2535. https://doi.org/10.1109/TKDE.2015.2416723
[8] Drew Branch. 2017. *Debugging iOS Applications: A Guide to Debug Other Developers' Apps.*
[9] Robert Brotzman, Danfeng Zhang, Mahmut T. Kandemir, and Gang Tan. 2021. Ghost Thread: Effective User-Space Cache Side Channel Protection. In *CODASPY '21: Eleventh ACM Conference on Data and Application Security and Privacy, Virtual Event, USA, April 26-28, 2021.* 233–244. https://doi.org/10.1145/3422337.3447846
[10] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. 2014. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.* 1037–1052.
[11] Yimin Chen, Xiaocong Jin, Jingchao Sun, Rui Zhang, and Yanchao Zhang. 2017. POWERFUL: Mobile app fingerprinting via power analysis. *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications* (2017), 1–9.
[12] Jack Cook, Jules Drean, Jonathan Behrens, and Mengjia Yan. 2022. There's always a bigger fish: a clarifying analysis of a machine-learning-assisted side-channel attack. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022.* 204–217. https://doi.org/10.1145/3470496.3527416
[13] Hoang Anh Dau, Anthony J. Bagnall, Kaveh Kamgar, Chin-Chia Michael Yeh, Yan Zhu, Shaghayegh Gharghabi, Chotirat Ann Ratanamahatana, and Eamonn J. Keogh. 2019. The UCR time series archive. *IEEE CAA J. Autom. Sinica* 6, 6 (2019), 1293–1305. https://doi.org/10.1109/jas.2019.1911747
[14] Christina Delimitrou and Christos Kozyrakis. 2017. Bolt: I Know What You Did Last Summer... In The Cloud. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017.* 599–613. https://doi.org/10.1145/3037697.3037703
[15] Angus Dempster, François Petitjean, and Geoffrey I. Webb. 2020. ROCKET: exceptionally fast and accurate time series classification using random convolutional kernels. *Data Min. Knowl. Discov.* 34, 5 (2020), 1454–1495. https://doi.org/10.1007/s10618-020-00701-z
[16] Angus Dempster, Daniel F. Schmidt, and Geoffrey I. Webb. 2020. MINIROCKET: A Very Fast (Almost) Deterministic Transform for Time Series Classification. *CoRR* abs/2012.08791 (2020). arXiv:2012.08791
[17] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. 2016. No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing

Analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. 414–432. https://doi.org/10.1109/SP.2016.32

[18] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. 2016. No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis. *2016 IEEE Symposium on Security and Privacy (SP)* (2016), 414–432.

[19] M. Abdelaziz Elaabid, Olivier Meynard, Sylvain Guilley, and Jean-Luc Danger. 2010. Combined Side-Channel Attacks. In *Information Security Applications - 11th International Workshop, WISA 2010, Jeju Island, Korea, August 24-26, 2010, Revised Selected Papers*. 175–190. https://doi.org/10.1007/978-3-642-17955-6_13

[20] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1-3 (2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015

[21] Hassan Ismail Fawaz, Benjamin Lucas, Germain Forestier, Charlotte Pelletier, Daniel F. Schmidt, Jonathan Weber, Geoffrey I. Webb, Lhassane Idoumghar, Pierre-Alain Muller, and François Petitjean. 2020. InceptionTime: Finding AlexNet for time series classification. *Data Min. Knowl. Discov.* 34, 6 (2020), 1936–1962. https://doi.org/10.1007/s10618-020-00710-y

[22] Yansong Gao, Jianrong Yao, Lihui Pang, Wei Yang, Anmin Fu, Said F. Al-Sarawi, and Derek Abbott. 2022. MLMSA: Multi-Label Multi-Side-Channel-Information enabled Deep Learning Attacks on APUF Variants. *CoRR* abs/2207.09744 (2022). https://doi.org/10.48550/arXiv.2207.09744 arXiv:2207.09744

[23] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. 2016. ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 1626–1638. https://doi.org/10.1145/2976749.2978353

[24] Mark A. Hall and Lloyd A. Smith. 1999. Feature Selection for Machine Learning: Comparing a Correlation-Based Filter Approach to the Wrapper. In *Proceedings of the Twelfth International Florida Artificial Intelligence Research Society Conference, May 1-5, 1999, Orlando, Florida, USA*. 235–239.

[25] Suman Jana and Vitaly Shmatikov. 2012. Memento: Learning Secrets from Process Footprints. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. 143–157. https://doi.org/10.1109/SP.2012.19

[26] Chia-Chi Lin, Hongyang Li, Xiao-yong Zhou, and XiaoFeng Wang. 2014. Screenmilker: How to Milk Your Android Screen for Secrets. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*.

[27] Jason Lines, Sarah Taylor, and Anthony J. Bagnall. 2016. HIVE-COTE: The Hierarchical Vote Collective of Transformation-Based Ensembles for Time Series Classification. In *IEEE 16th International Conference on Data Mining, ICDM 2016, December 12-15, 2016, Barcelona, Spain*. 1041–1046. https://doi.org/10.1109/ICDM.2016.0133

[28] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. 2017. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II*. 191–209. https://doi.org/10.1007/978-3-319-66399-9_11

[29] Oleksiy Lisovets, David Knichel, Thorben Moos, and Amir Moradi. 2021. Let's Take it Offline: Boosting Brute-Force Attacks on iPhone's User Authentication through SCA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021, 3 (2021), 496–519. https://doi.org/10.46586/tches.v2021.i3.496-519

[30] Wei Liu, Youwei Zhang, Yonghe Tang, Huanwei Wang, and Qiang Wei. 2023. ALScA: A Framework for Using Auxiliary Learning Side-Channel Attacks to Model PUFs. *IEEE Trans. Inf. Forensics Secur.* 18 (2023), 804–817. https://doi.org/10.1109/TIFS.2022.3227445

[31] Benjamin Lucas, Ahmed Shifaz, Charlotte Pelletier, Lachlan O'Neill, Nayyar Abbas Zaidi, Bart Goethals, François Petitjean, and Geoffrey I. Webb. 2019. Proximity Forest: an effective and scalable distance-based classifier for time series. *Data Min. Knowl. Discov.* 33, 3 (2019), 607–635. https://doi.org/10.1007/s10618-019-00617-3

[32] Yan Michalevsky, Aaron Schulman, Gunaa Arumugam Veerapandian, Dan Boneh, and Gabi Nakibly. 2015. PowerSpy: Location Tracking Using Mobile Device Power Analysis. *ArXiv* abs/1502.03182 (2015).

[33] George A Miller. 1995. WordNet: a lexical database for English. *Commun. ACM* 38, 11 (1995), 39–41.

[34] Xuan Vinh Nguyen, Julien Epps, and James Bailey. 2010. Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance. *J. Mach. Learn. Res.* 11 (2010), 2837–2854. https://doi.org/10.5555/1756006.1953024

[35] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. 1406–1418. https://doi.org/10.1145/2810103.2813708

[36] Alejandro Pasos Ruiz, Michael Flynn, James Large, Matthew Middlehurst, and Anthony J. Bagnall. 2021. The great multivariate time series classification bake off: a review and experimental evaluation of recent algorithmic advances. *Data Min. Knowl. Discov.* 35, 2 (2021), 401–449. https://doi.org/10.1007/s10618-020-00727-3

[37] Walter J. Scheirer, Anderson de Rezende Rocha, Archana Sapkota, and Terrance E. Boult. 2013. Toward Open Set Recognition. *IEEE Trans. Pattern Anal. Mach. Intell.* 35, 7 (2013), 1757–1772. https://doi.org/10.1109/TPAMI.2012.256

[38] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. 2018. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*.

[39] Claude E. Shannon. 1948. A mathematical theory of communication. *Bell Syst. Tech. J.* 27, 3 (1948), 379–423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x

[40] Ahmed Shifaz, Charlotte Pelletier, François Petitjean, and Geoffrey I. Webb. 2020. TS-CHIEF: a scalable and accurate forest algorithm for time series classification. *Data Min. Knowl. Discov.* 34, 3 (2020), 742–775. https://doi.org/10.1007/s10618-020-00679-8

[41] Connor Shorten and Taghi M. Khoshgoftaar. 2019. A survey on Image Data Augmentation for Deep Learning. *J. Big Data* 6 (2019), 60. https://doi.org/10.1186/s40537-019-0197-0

[42] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2019. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 639–656.

[43] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. 2018. ProcHarvester: Fully Automated Analysis of Procfs Side-Channel Leaks on Android. *Proceedings of the 2018 on Asia Conference on Computer and Communications Security* (2018).

[44] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. 1–9. https://doi.org/10.1109/CVPR.2015.7298594

[45] Chang Wei Tan, Angus Dempster, Christoph Bergmeir, and Geoffrey I. Webb. 2022. MultiRocket: multiple pooling operators and transformations for fast and effective time series classification. *Data Min. Knowl. Discov.* 36, 5 (2022), 1623–1646. https://doi.org/10.1007/s10618-022-00844-1

[46] Shruti Tople and Prateek Saxena. 2017. On the Trade-Offs in Oblivious Execution Techniques. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*. 25–47. https://doi.org/10.1007/978-3-319-60876-1_2

[47] Qingsong Wen, Liang Sun, Fan Yang, Xiaomin Song, Jingkun Gao, Xue Wang, and Huan Xu. 2021. Time Series Data Augmentation for Deep Learning: A Survey. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*. 4653–4660. https://doi.org/10.24963/ijcai.2021/631

[48] Qiuyu Xiao, Michael K. Reiter, and Yinqian Zhang. 2015. Mitigating Storage Side Channels Using Statistical Privacy Mechanisms. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. 1582–1594. https://doi.org/10.1145/2810103.2813645

[49] Jiadi Yu, Li Lu, Yingying Chen, Yanmin Zhu, and L. Kong. 2021. An Indirect Eavesdropping Attack of Keystrokes on Touch Screen through Acoustic Sensing. *IEEE Transactions on Mobile Computing* 20 (2021), 337–351.

[50] Kehuan Zhang and XiaoFeng Wang. 2009. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*. 17–32.

[51] Li Zhang, Parth H Pathak, Muchen Wu, Yixin Zhao, and Prasant Mohapatra. 2015. Accelword: Energy efficient hotword detection through accelerometer. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. 301–315.

[52] Nan Zhang, Kan Yuan, Muhammad Naveed, Xiao yong Zhou, and Xiaofeng Wang. 2015. Leave Me Alone: App-Level Protection against Runtime Information Gathering on Android. *2015 IEEE Symposium on Security and Privacy* (2015), 915–930.

[53] Xiaokuan Zhang, Xueqiang Wang, Xiaolong Bai, Yinqian Zhang, and XiaoFeng Wang. 2018. OS-level Side Channels without Procfs: Exploring Cross-App Information Leakage on iOS. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*.

[54] Man Zhou, Qian Wang, Jingxiao Yang, Qi Li, Feng Xiao, Zhibo Wang, and Xiaofeng Chen. 2018. PatternListener: Cracking Android Pattern Lock Using Acoustic Signals. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018).

[55] Xiao-yong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A. Gunter, and Klara Nahrstedt. 2013. Identity, location, disease and more: inferring your secrets from android public resources. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. 1017–1028.