

Towards Automatic Detection of Nonfunctional Sensitive Transmissions in Mobile Applications

Hao Fu ¹, Pengfei Hu, Zizhan Zheng ², *Member, IEEE*, Aveek K. Das, Parth H. Pathak, Tianbo Gu, Sencun Zhu, and Prasant Mohapatra ³, *Fellow, IEEE*

Abstract—While mobile apps often need to transmit sensitive information out to support various functionalities, they may also abuse the privilege by leaking the data to unauthorized third parties. This makes us question: *Is the given transmission required to fulfill the app functionality?* In this paper, we make the first attempt to automatically identify suspicious transmissions from app visual interfaces, including app names, descriptions, and user interfaces. We design and implement a novel framework called `FlowIntent` to detect nonfunctional transmissions at both software and network levels. During the exercising of the given apps, `FlowIntent` automatically detects privacy-sharing transmissions and determines their purposes by utilizing the fact that mobile users rely on visible app interface to perceive the functionality of the app at certain context. The characterizations of nonfunctional network traffic are then summarized to provide network level protection. `FlowIntent` not only reduces the false alarms caused by traditional taint analysis, but also captures the sensitive transmissions missed by widely-used taint analysis system `TaintDroid`. Evaluation using 2125 sharing flows collected from more than a thousand running instances shows that our approach achieves about 94 percent accuracy in detecting nonfunctional transmissions.

Index Terms—Security and privacy, information flow controls, mobile code security, network-level security and protection

1 INTRODUCTION

SMART phones are becoming indispensable to many of us, thanks to the rich functionalities provided by a large number of mobile applications (or apps, for short). The sheer number of these apps, however, makes it challenging to understand their behavior and control their quality before publishing. On one hand, these apps collect sensitive data, such as locations, contacts and phone identifiers, to support various functionalities. For example, a weather app queries user's locations to provide precise humidity information. On the other hand, they may also abuse the resources and share these data for purposes irrelevant to app functionality, e.g., analytics or advertisement. Therefore, it is crucial to automatically infer the intention of sensitive transmissions initiated by mobile apps and only signal an alarm when a suspicious transmission is located.

Most existing solutions [4], [9], [14], [51], [52] treat *all* sensitive transmissions as suspicious, which, however, inevitably

produce many false alarms as users often tolerate transmissions of private data when they are used for promised services [8], [45], [54]. As suggested in [45], a better approach is to *determine whether the transmissions are used to fulfill the underlying app functionalities*, and consider the extraneous transmissions such as those used for advertising, analytics, cross-application profiling and social computing, as threats. For instance, a traffic flow that carries users location information for driving navigation should be treated as benign and blocking it will cause app malfunction. But a location-sharing request triggered by a flash light app is suspicious and should be alarmed.

However, locating nonfunctional sensitive transmissions is far from trivial. First, it is very difficult if not impossible for machines to understand the functionalities of apps since the code stream itself does not directly unveil its semantics. Second, catching up the evolving speed of adversarial techniques is hard. The level of code obfuscation have changed dramatically since the first discovery of Android malware in 2010 [44]. Malicious developers also continuously update the addresses of their servers to avoid detection [57]. Last but not least, although it is desired to involve as little human intervention to recognize unintended transmissions as possible, it is almost impossible to have an completely automated method because of the complex nature of user intention [54].

To date, various methods have been proposed to detect and isolate the third-party libraries that may incur privacy threats [6], [26], [34], [41], [46], [56]. However, they either rely on the namespace and the program structure [6], [26], [56], thus suffering from the evasion attacks such as obfuscation and call graph manipulation, or count on the deep cooperation of developers [34], [41], [46], which ignores a

- H. Fu, T. Gu, and P. Mohapatra are with the Department of Computer Science, University of California Davis, Davis, CA 95616 USA. E-mail: {haofu, tbg, pmohapatra}@ucdavis.edu.
- P. Hu is with the School of Computer Science and Technology, Shandong University, Jinan 250100, China. E-mail: phu@sdu.edu.cn.
- Z. Zheng is with the Department of Computer Science, Tulane University, New Orleans, LA 70118 USA. E-mail: zzheng3@tulane.edu.
- A.K. Das is with Forescout Technologies, San Jose, CA 95134 USA. E-mail: akdas@ucdavis.edu.
- P.H. Pathak is with the Computer Science Department, George Mason University, Fairfax, VA 22030 USA. E-mail: parth.pathak@gmail.com.
- S. Zhu is with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802 USA. E-mail: szhu@cse.psu.edu.

Manuscript received 5 July 2019; revised 7 Mar. 2020; accepted 22 Apr. 2020.
Date of publication 4 May 2020; date of current version 31 Aug. 2021.
(Corresponding author: Pengfei Hu)
Digital Object Identifier no. 10.1109/TMC.2020.2992253

great deal of intentional data leakage driven by under-the-table income. More importantly, all of them are designed to handle the misbehavior from isolated ad libraries, and they do not apply to malicious transmissions embedded in the core app logic.

As shown in several recent user studies [31], [49], [50], it is crucial to account for the *context* pertinent to sensitive resource accesses to enable effective resource access control. In particular, it is observed that the security decision of a user is strongly correlated with the foreground app and the visibility of the requesting app (i.e., the app that is requesting the sensitive resource, which can be different from the foreground app). This is because users often rely on displayed information to infer the intention of a request and tend to deny requests that appear irrelevant to app's functionalities [49]. From another perspective, apps leverage UIs to help bridge the gap between functionality and behavior. As indicated in AppFence [21], resource accesses that do not fulfill functionalities can be greatly purged without causing influential UI side effects. Inspired by these observations, we propose a novel approach that mimics the decision making process of users to automatically identify sensitive transmissions not indicated by app functionality. Our approach actively infers the underlying functionality provided by a sensitive transmission from visible app information, including app name, description, and user interface (UI), and alarm those transmissions that cannot be justified. In particular, app name and description provide general information about the core functionality of an app, while user interface helps further identify more detailed and context-dependent functionality of an app.

We implement *FlowIntent*, a proof-of-concept framework composed of two main modules: *AppInspector* and *TrafficAnalyzer*. *AppInspector* first automatically launches and exercises a given app, with corresponding app context and sensitive transmissions recorded. The nonfunctional transmissions are then identified with the help of machine learning, utilizing the visible app information including app name, description and the UI for the running period. With the set of transmissions labeled by *AppInspector*, *TrafficAnalyzer* then constructs classifiers to localize anomalies at the network traffic level by using flow level features *only*.

Our design provides comprehensive detection at both the app level and the network level. At the app level, *AppInspector* can be used as an automatic app auditing service for app markets and allows app market operators to identify privacy leakages in an app before releasing it to the market. As previous research [15], [45], [52] has pointed out, even authentic apps available in popular Android app stores may put users at risk by stealthily sending users' private information out in a user-unexpected way. Unlike existing program analysis techniques [4], [9], [14] that report every sharing transmission and thus generate massive false alarms, the automatic intention inference engine provided by *AppInspector* enables market operators to only focus on suspicious transmissions not required by app functionality. In addition, by leveraging human-readable features, our approach is resilient to code obfuscation, a key limitation of the existing namespace-based approaches [6], [26], [47]. Moreover, *FlowIntent* does not require any modification of app code.

At the network level, *TrafficAnalyzer* can be deployed on Network Intrusion Detection Systems (NIDS) or Access Points (AP) to prevent unexpected leakages over HTTP(S) flows on-the-fly. *TrafficAnalyzer* attempts to fill the missing part of existing network-based privacy-leakage detection work [38], which do not determine what information needs to be shared to fulfill the underlying app functionality. It can also be used in app audit services to detect nonfunctional flows before installation. We remark that *FlowIntent* goes beyond the traditional traffic-based approach that depends on a set of suspicious URLs [11], [27], [37], which often requires significant human effort and is especially challenging for mobile platforms since a large number of apps are normal apps that only leak private data occasionally. In addition, manually generated lists of suspicious URLs can barely keep pace with the fast growth of app markets, where new network addresses are continuously mushrooming. Further, it is often difficult to tell even for humans whether a flow is targeting an unexpected destination simply from the URL. This is especially true when the illegal flows share the same domain with legal flows. To this end, our approach relies on machine learning to largely relax the dependence on human intervention, which enables fast generation of signatures for new traffic patterns. We emphasize that the objective of *TrafficAnalyzer* is to identify nonfunctional leaking flows instead of leaking apps. The app level information is only used to help collect a comprehensive list of suspicious flows for training, and the network signatures generated by *TrafficAnalyzer* are solely based on the characteristics of HTTP(S) flows. Therefore, NIDS does not need to know which app has generated a flow in order to identify privacy leakage in the flow. Our work is orthogonal to existing works that focus on learning app identity from network traffic [12].

In summary, our main contributions are as follows:

- We propose a novel definition of privacy leakage based on visible app interfaces that bridges the gap between app functionality and flow level behavior in mobile platforms.
- We design and implement a cross-layer framework for detecting privacy leakage in mobile platforms at both program and network levels. Our approach is easy to deploy and can potentially adapt to the constantly evolving leakage patterns. We further show how our novel design is capable to detect the malicious transmissions hidden behind legitimate app context.
- Our approach achieves about 96 percent accuracy on 1,666 privacy-sharing contexts and 94 percent accuracy on 2,125 leaking HTTP flows. Moreover, evaluation shows that our approach is able to identify leakages missed by dynamic taint analysis.

This manuscript is an extension of the conference version [16] and presents a more accurate and detailed description of the problem definition, system design, implementation, and evaluation. More concretely, we have added a discussion on the most recent studies published after our conference paper to better motivate our work. *FlowIntent* previously focused on the front window of the given apps. We have expanded it by considering more windows using a smarter UI exerciser to locate deeper leaks. While the conference paper only discusses

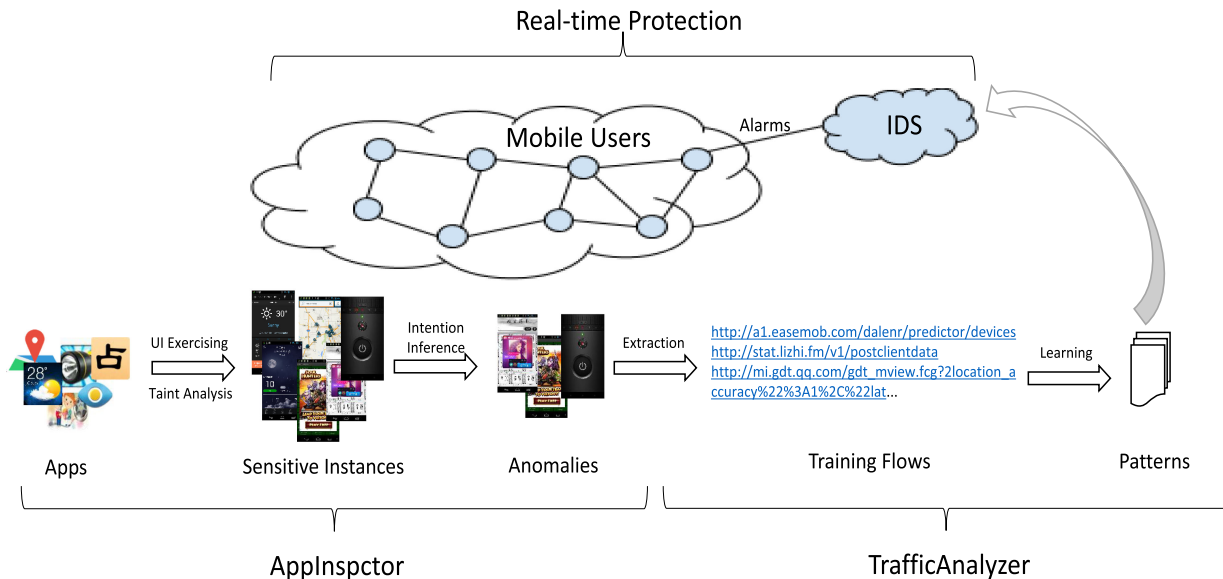


Fig. 1. FlowIntent system architecture.

the results about location leakages, we have extended it by considering more types of personal information leaks. We have also collected more samples from other data sources and redone the relevant experiments. We have further added an important experiment to show that FlowIntent is capable to detect stealthy nonfunctional transmissions hidden behind legal context. With a more comprehensive analysis, we noticed that one-class SVM used before is too sensitive to its parameters and we replaced it by a more mature and practical algorithm for our setting.

The rest of the paper is organized as follows. We present the threat model in Section 2, followed by an overview of the system in Section 3. We illuminate the design and implementation of app context modeling and traffic learning schemes in Sections 4 and 5, respectively. After presenting the evaluation results in Section 6, we list related work in Section 7. We further discuss the future work in Section 8 and conclude our paper in Section 9. The code is open-source and publicly available at: <https://pmlab.cs.ucdavis.edu/flowintent>.

2 THREAT MODEL

We target threats from the transmission flows generated by third-party mobile apps, which carry sensitive personal or device information that does not provide intended functionalities to users. The flows are constructed either by intended malicious logic embedded in apps, or malicious libraries that piggyback on vulnerable apps. Functional information sharing may also be exploited for illegal purposes, which remains an open problem and we do not consider that in this work. We assume that the platforms where FlowIntent is deployed, including the underlying mobile operating system of AppAnalyzer and the NIDS with TrafficAnalyzer integrated, are trustworthy and uncompromised.

3 SYSTEM OVERVIEW

In this section, we first describe the threat model, and then present a high-level overview of our approach for detecting nonfunctional sensitive data transmissions.

Fig. 1 gives an overview of FlowIntent, which has two key modules and works as follows:

AppInspector: From the set of apps we collected, we first remove those not requesting any sensitive permission of our interest. We then identify the running instances that transmit out personal or device information with automatic UI exercising and dynamic taint analysis. For each detected sensitive transmission, we store its app level contextual data (app name, description and UI) and the captured correspondent traffic flows. We then construct machine learning classifiers to identify sensitive transmissions that do not fulfill the functionalities indicated by their app contexts. The collected suspicious network flows are further fed to the next stage.

TrafficAnalyzer. Given the nonfunctional HTTP flows labeled in the last phase and the pre-collected normal flows, we derive both statistical and lexical features to classify unseen traffic on-the-fly. The statistical features are lightweighted and can be obtained without deeper inspection, but they do not contain any semantic information. The rich semantic data are encoded into lexical features, which require

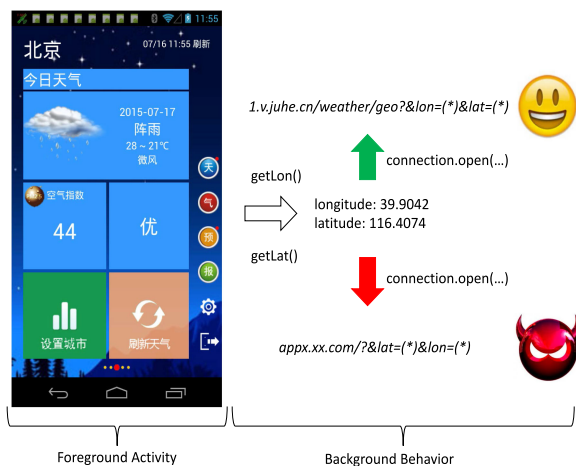


Fig. 2. Stealthy harmful transmission hidden behind normal location-sharing context.




App	Context	Source	Extracted	Preprocessed
	Name	apk meta-data	"LocalWeather"	n_local, n_weather
	Description	App market	"Accurate, simple & fast local weather forecast. Powered by The Dark Sky Forecast API. Stay tuned, updates will come soon!"	d_weather
	UI	Auto Exerciser Dynamic Tracking		dalvik, iceland, 9, °, cloudy, humidity, 7, day ...
				week, weather, °, monday, tuesday, wednes...
			1, am, 2, 9, °, overcast, 3, cloudy, 4, 5, 6, ...	

Fig. 3. Example app context that includes the app name n (from apk meta-data), the description d (from app market) and the set of runtime user interfaces W (recorded via auto exerciser). The corresponding preprocessed results for each field are shown in the last column.

more computational power to get and may be intractable when the flows are encrypted.

In addition to supervised learning that requires both non-functional and benign flows, *TrafficAnalyzer* is also able to perform analysis solely on nonfunctional sensitive flows. Based on the similarities among the target flows, *TrafficAnalyzer* clusters them into subgroups, and derive the network signature, in the form of regular expression, from every group.

We observe that some harmful network flows are hidden behind functional ones with proper app contexts shown to users. For example, Fig. 2 is a screenshot taken from an authentic weather app in a prevailing Chinese app market. Based on the user interface, even a cautious end user could allow the app to send location information out of device since it is helpful to offer precise weather reports. Unfortunately, this weather app also collects user's location data for illegal purposes. As *AppInspector* mimics the user decision making process and solely examines the visible content, it will not report any misbehavior similar to this. Fortunately, as we will show later in Section 6, the patterns derived from *TrafficAnalyzer* can be utilized to locate nonfunctional transmission missed by *AppInspector*. The beneficial connection between *AppInspector* and *TrafficAnalyzer* provides a more complete protection.

4 APPINSPECTOR

As described earlier, *AppInspector* leverages app context of each sensitive transmission to mark nonfunctional instances. Below we first give definitions of sensitive transmissions and app context used in this work.

Definition 1. Sensitive Transmission *A transmission t is sensitive if it carries the data obtained from a protected resource $r \in R$.*

We note that what kind of resource needs to be protected is application and user dependent. At the current stage, we think the heuristic choices specified by *TaintDroid* and *FlowDroid* [4] is reasonable and comprehensive.

Definition 2. App Context *Given a sensitive transmission t generated by an application a , its app context is a tuple*

$c_t = \langle n, d, W \rangle$, where (1) n is the name of a shown to the user; (2) d is the description of a available on the market and its value could be empty in case of no associated description; (3) W is the set of runtime user interfaces of a right after t is triggered.

Fig. 3 gives an example of app context extracted from a weather forecast app, which transfers device GPS data to a service provider.

We are now ready to define the semantic relationship between t and c_t with respect to the underlying app functionalities.

Definition 3. Nonfunctional Sensitive Transmission *A sensitive transmission t is considered nonfunctional if it is not required in any $f \in F$, where F is the set of app functionalities indicated by the app context c_t .*

Our definition of nonfunctional sensitive transmissions offers a novel angle to understand privacy leakage. However, due to the complexity of app's logic and the potentially infinite number of implementations, it is next to impossible to identify a finite set of formal rules to determine whether a transmission is nonfunctional. Therefore, instead of manually specifying the mapping between transmissions and functionalities and that between functionalities and context, we directly bridge the semantic gap between a sensitive transmission t and its associated context c_t through *machine learning*. By categorizing sensitive transmissions based on app context, we formulate the detection of nonfunctional transmissions as a classification problem. In the following subsections, we first describe how we build the initial data set to train our machine learning models. The details of the learning process are then discussed in Section 4.4. We present the evaluation results in Section 6.1.

4.1 Collecting Sensitive Running Instances

Collecting sensitive transmissions and their foreground context is not a trivial task. Although app name n can be easily extracted from its package meta-data and app description d can be crawled from the corresponding app store page, it is challenging to automatically trigger the sensitive behaviors and correctly render their foreground user

interfaces. The existing testing tool provided by Android can potentially help collect app context we need. However, it is not implemented for security analysis so that its fuzzing engine would waste time on many random events that are irrelevant to our objective. In this paper, we integrate TaintDroid [14] with COSMOS [17] to automatically collect sensitive running instances of apps, where a sensitive running instance is a pair (t, W) presenting the set of foreground interfaces after initiating t . More specifically, we install the target app inside a TaintDroid sandbox. The sandbox has `tcpdump`¹ installed to collect the TCP packets generated during examination. TaintDroid provides a powerful real-time tracking system that alarms the end user whenever a PII-sharing HTTP(S) flow is triggered. However, it can neither trigger sensitive calls automatically nor collect contextual data. To this end, we use COSMOS [17] to interact with the app to automatically trigger the internal API calls of the app that access the concerned resources. COSMOS contains a smart hybrid program analysis framework that leverages over-approximated static analysis to identify potential sensitive API invocations, and attempt to activates the calls by properly awaking their front-end window. Specifically, it first identifies permission-protected API calls through method signatures and constructs a call graph for the given app. The set of call graph entry points of the sensitive API calls and the set of widgets that invoke the entry points are then identified by carefully traversing through the call graph. For each Activity window recognized by static analysis, COSMOS then attempts to trigger it by properly sending inter-component messages during runtime. We wait for a few seconds² after each invocation and if there are sensitive HTTP(S) flows reported by TaintDroid, COSMOS will take a screenshot of the app and export the foreground window into a hierarchy XML file, which is a standard format to represent the layout of user interface in Android. We currently do not consider the more complicated multi-window scenario and leave that for future study. For the captured images of foreground windows, FlowIntent can leverage Optical Character Recognition (OCR) [20] to extract the embedded text in order to capture all the context information. In summary, a transmission reported by TaintDroid (with the TCP packets stored by `tcpdump`) and the window extracted by COSMOS together forms a sensitive running instance. Combining it with the app name and description, we now obtain the complete app context $c_t = \langle n, d, W \rangle$ for every identified sensitive transmission t .

4.2 Labeling App Context

We classify every app context used for training and testing into two classes and manually label them based on the nature of the underlying transmission. We consider a context $c_t = \langle n, d, W \rangle$ *legitimate* if based on our understanding, at

1. <http://www.tcpdump.org/>

2. We followed the rule of thumb threshold used in MadFroud, in which they “create a new emulator image, install the app on the new emulator, run the app in the foreground for 60 seconds, put the app into the background, and run for another 60 seconds” to capture the ad and analytical traffic [11]. Similarly, we triggered each Activity window that may lead to sensitive flows and waited around 60 seconds. From our observation, 60 seconds is generally sufficient to retrieve the useful flow data.

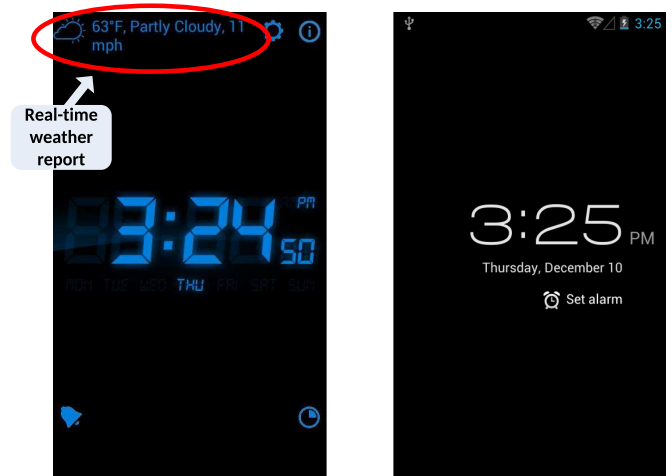


Fig. 4. The screenshots of two running alarm clock apps.

least one of the indicated functionalities may require sending the resource r carried by t out of device; otherwise it is treated as *illegitimate*. In particular, we first note that developers often name apps according to their functionalities. This is especially true for apps in the “tools” category in app stores, which are usually named based on their core functionalities. For instance, we expect that the app “LocalWeather” to be a weather app and “SuperLed” to be a flashlight app. Nevertheless, not all apps have meaningful names that imply their functionality. The names such as “Yelp”, “Uber” and “Lyft” by themselves do not indicate the services provided by the apps. Fortunately, an app description, when it exists, gives us a more fine-grained explanation of the app’s core functionality. A sentence such as “the app uses your location so your driver knows where to pick you up” [2] explicitly tells us that this app transmits users’ locations to match nearby drivers. However, both app names and app descriptions only provide high-level static information of apps. To capture the dynamic fine-grained behavior of an app, we further collect user interfaces that are especially useful when (1) the app has multiple functionalities that are valid under different contexts, and (2) some functionalities provided are not clearly mentioned in the app description. For instance, from the name and the description of an alarm clock app, one may conclude that a location-sharing flow generated from the app is suspicious since alarm apps usually do not need location information to fulfill their functionality. Although this is typically true, there are exceptions. As an example, consider the two windows shown in Fig. 4. From the screenshots we can infer that the app on the right provides a functionality of “alarm clock”, while the app on the left provides both “alarm clock” and “weather” functionalities. Suppose both apps collect user’s location under the displayed windows. The former context is then treated as illegitimate since location information is irrelevant to “alarm clock”, while the latter context is considered legitimate due to the fact that “weather” requires location data to provide an accurate report.

Our dataset mainly involves the transmissions of *device location* from a networking or GPS provider, and *device identifier* such as IMEI and ICCID. Similar to [47] where prevalent functionalities identified from method names and variable names in Java code are given, we summarize the core functionalities observed from app context regarding the two

TABLE 1
The Functionalities Supported by the Sensitive Transmissions

Resource	Functionality
Location	map, navigation, weather, news, nearby service, anti-theft
Dev id	anti-theft, fraud prevention

main types of sensitive transmissions in Table 1. “Nearby service” in the table is a generic name of many location-based services (LBS), such as meal delivery, ride sharing, pest inspection, etc. Compared to device location, phone id has much less functionalities. In particular, “anti-theft” leverages both phone id and location to provide real-time device tracking. Commodity apps may also leverage device identifiers in fraud prevention efforts [2]. We currently treat all user authentication pages as legal contexts for device id transmission. Due to the page limit, we refer to [47] for a more detailed description of other functionalities.

4.3 Preprocessing

With the labeled sample pairs (r, c_t) , we then leverage machine learning to automatically learn the semantic mapping between the resource type of the transmissions and the app context, which can be used to identify unseen anomalies in the future. Before feeding the samples to the learning algorithms, we need to preprocess the raw data and convert them into feature values. Fig. 3 gives an example of the preprocessing outcome. We will discuss them in detail, starting from app names.

We extract the app names from the apk meta-data. We do not simply treat each name as a feature. Instead, we extract popular words that frequently appear in app names with the help of an existing word list.³ For example, “local” and “weather” are extracted from the app name “LocalWeather”. We also add some new words to the list such as “tech” and “nav”, which are used by some apps as abbreviations.

We follow the approach in [19] to map app descriptions into topics, which provides a concise representation of the main functionalities of apps. We first utilize the Natural Language Toolkit (NLTK)⁴ to tokenize English sentences in a description into words. The words are then fed to a NLTK’s stemmer, where they are reduced to their root forms. We also remove all the stop words. We do the similar things for apps with Chinese descriptions by using Jieba.⁵ We then apply text mining to get the most related topics. Since detailed topic modeling is beyond the scope of this paper, we directly leverage the set of keywords for each corresponding topic given in [19]. For a description that includes keywords belonging to different topics, we choose the top topic that is hit by the maximum number of different keywords. It is possible that a description does not fall into any topics generated by [19]. We assign such an app a coarse topic given by the corresponding app market. Each topic is treated as a single feature in our learning model. Table 2 shows some examples of topics given in CHABADA [19],

TABLE 2
Sample Topics

CHABADA	personalize, games and cheat sheets, music, navigation and travel, language, share, health, kids, ringtones and sound, search and browse
Google Play	sports, social, shopping, productivity, tools, photography, personalizing, medical, lifestyle, finance, libraries and demo, music and audio
Baidu App Market	social and communication, system and tools, finance and shopping, themes, photography, video and audio, lifestyle, office, books

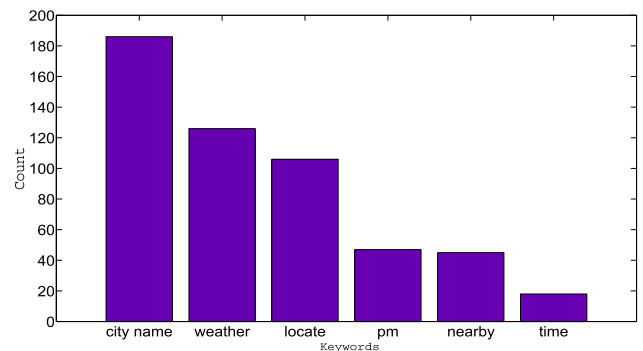


Fig. 5. The top popular keywords (4808 keywords in total) of UIs collected from 634 legal location-sharing instances.

Google Play,⁶ and Baidu App Market.⁷ Table 3 shows the keywords associated with some topics.

Unlike an app description that is typically constructed with complete sentences, the content of a UI component is often composed of phrases or short and incomplete sentences. Therefore, it is possible to achieve accurate classification without using sophisticated natural language processing (NLP) techniques that are needed to preserve the order of tokens. In this work, we utilize the simple yet powerful bag-of-words technique that is commonly used in spam detection [10] by treating each distinct word appeared in the window of a running instance as a separate (binary) feature. In addition to pure text data, we have introduced some extra features that can help capture app functionalities. For example, we create a binary feature “city-clickable” to represent a special type of clickable widgets that have a city name and are located at the top left of the windows. This feature is commonly found in instances that legally take user’s location to deliver local news or services and users can change their regions by clicking that button. Fig. 5 shows the set of most popular tokens that appear in the UI of legal location-sharing applications, where “city name” includes all the concrete city names, and “pm” indicates either particulate matter or post meridiem. We observe that these keywords closely match our intuition about location related instances. In particular, “city name”, “locate” and “nearby” are directly related to locations, while “weather” and “pm” appear in weather reporting instances, which are typically location sensitive. Moreover, many location related services have a small widget that shows the current time, which explains why “time” is also a popular keyword.

3. <https://github.com/first20hours/google-10000-english>

4. <http://www.nltk.org/index.html>

5. <https://github.com/fxsjy/jieba>

6. <https://play.google.com/store/apps>

7. <http://shouji.baidu.com/>

To properly separate the three types of features, we attach the prefix “n_” and “d_” to the key words extracted in app name and description, respectively. Therefore, a word like “weather” that appears in multiple feature spaces will not interfere each other, as shown in Fig. 3. In addition to app context related features, we also add the resource type (e.g., “r_location” and “r_id”) as a feature to clearly specify the nature of the instance.

4.4 Voting

The labeled app context will be used to further help classify traffic flows, which will be elaborated in the next section. Therefore, it is important to ensure both high precision and recall at this stage. To this end, we consider three well-studied learning algorithms, *random forest*, *SVM* and *logistic regression*, and adopt a commonly used consensus voting approach [7] to filter potential misclassified instances. That is, only the set of instances where all the three algorithms give the same classification results are retained. We provide the evaluation results of our context classification in Section 6.1.

4.5 App Audit

Thus far, we have described the main structure of AppInspector. After training, AppInspector can automatically expose the sensitive transmissions and identify the nonfunctional ones that are not supported by their app context. Hence, AppInspector itself can be deployed as a standalone service to help app market operators identify the suspicious transmissions inside a newly uploaded app, before releasing it to public. However, although contextual information indicates what kind of transmission is appropriate, it alone is not capable to precisely tell whether a specific flow is functional or not. As we mentioned in Section 3, nonfunctional flows may also exist behind the app context that semantically match the resource type of the flows. We can adopt the patterns generated from TrafficAnalyzer to further recognize the stealthy transmissions as we discuss in the next section.

5 TRAFFICANALYZER

In this section, we describe the design of TrafficAnalyzer, which processes the traffic flows collected by AppInspector and build classifiers to detect nonfunctional flows. We consider two models: (i) Supervised learning model that summarizes the patterns for nonfunctional sensitive flows and normal flows; (ii) Clustering approach that generates the signatures for nonfunctional sensitive flows only.

5.1 Traffic Flows

Based on our definition, an illegal app context indicates that no underlying functionality requires the given sensitive transmission. Therefore, we treat all sensitive transmissions behind illegal contexts as nonfunctional.⁸ However, it is doubtful to straightforwardly label all flows behind legal contexts as functional. Although resource sharing is justified for these instances, it is crucial to detect the nonfunctional/malicious flows hidden behind as shown in Fig. 2. To get the ground truth of the flows behind legal contexts,

8. A malicious app may first send sensitive data to an inquiry service provider (e.g., Google), and then forward the feedback to a bad server. We will discuss this situation in Section 8.

the following steps are applied: (1) We first examine the destination hostname of the flow. If it belongs to popular service providers such as “map.google.com”, “amap.com” and “loc.baidu.com”, the flow is labeled as functional; (2) Otherwise we check the plain text content in the response, and the flow is considered functional if the response is related to the resource sent; (3) For the rest of flows that cannot be determined by above approaches, we have implemented a blocking approach as follows. For each of these flows, we first set firewall rules (based on the TaintDroid reports) to block the flow. We then clean the cache on the device, rerun the app, and observe the corresponding window. The flow is labeled as nonfunctional if nothing unusual is observed, indicating that the app’s functionality is not affected even if the flow is blocked.

5.2 Supervised Learning

As we do not know what features could be helpful beforehand for our supervised learning, we follow the feature lists that have been widely used in previous traffic classification systems [24], [27], [28], [37], [38] and let the learning algorithms choose from them based on their criteria.

Statistical Features. For each HTTP flow that forms a session and is identifiable by a 4-tuple (source IP, source port, destination IP, destination port), the following statistical features are calculated:

- Total number of TCP packets
- Total number of uplink TCP packets
- Total number of HTTP packets (Packets with HTTP application layer present)
- Packet size of all TCP packets
- Packet size of uplink TCP packets
- Packet size of downlink TCP packets
- Time interval between two consecutive TCP packets

The first three attributes hold single scalar values while the rest four attributes are distributions, each represented using 7 statistical features, namely, minimum, maximum, median, mean, standard deviation, skewness and kurtosis. While extracting the features from the traffic, no prior information about the shape of the distributions (Gaussian or not) are assumed. Since our primary concern is the accurate representation of distributions obtained from the data, similar to the network features used in [24], we consider the first four moments (mean, variance, skewness and kurtosis) in addition to maxima, median and minima. The total number of statistical features is 31.

Take the location-related flows as an example. Fig. 6(a) shows that non-sensitive flows generally have a significantly higher number of TCP packets as compared to location flows, which is expected, while nonfunctional location flows have a slightly lower packet count than functional flows. Fig. 6(b) shows that non-location flows have larger downlink packet sizes compared with location flows, which is mainly contributed by data-intensive apps. Among location-sharing flows, nonfunctional flows are typically ad flows that are usually responded with an advertisement, which makes the maximum packet size in the downlink traffic larger than those for functional flows, where the response is mainly control packets denoting that the location has been received. Fig. 6(c) shows that the functional flows have higher average

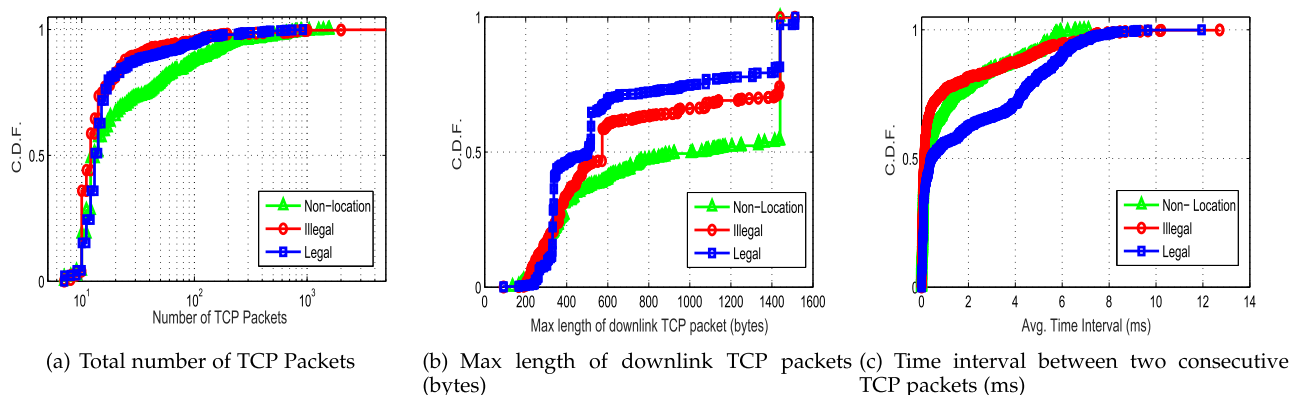


Fig. 6. CDFs of statistical features.

TABLE 3
Example Topics With Relevant Keywords [19]

“navigation and travel”	map, inform, track, gps, naving, travel, citi
“weather and stars”	weather, forecast, locate, temperatur, city, light
“health”	weight, bodi, exercise, diet, workout, medic

packet inter-arrival time, since the benign servers need more time to handle users’ specific location-related requests and generate appropriate responses.

Lexical Features. In addition to statistical features, we also consider lexical features derived from the textual properties of URLs, which often contains useful patterns to distinguish benign and malicious traffic [27]. The intuition is that URLs may contain words that can be used to differentiate the purposes of location requests. As an example, consider an illegitimate location sharing with the following URL: `ads.appsgeyser.com/?tlat=38.5&tlon=-121...`. We can see that the domain name `ads.appsgeyser.com` has a prefix of “ads”, which indicates the advertisement purpose of the request. As another example, consider a location-sharing flow generated by a weather forecast application with the URL `v.juhe.cn/weather/geo?&lon=-121.7&lat=38.5`. The path portion of the URL `weather/geo?&lon=-121.7&lat=38.5` includes the word “weather”, indicating that the server behind the URL is a weather information provider. Moreover, both URLs contain exact longitude and latitude values in the plain text, which can be used to identify location-sharing flows from all outgoing traffic traces. We follow the “bag-of-words” approach used in [27] and treat each token inside a URL as a binary feature. Below is a list of lexical features considered.

- Binary feature for each token in the host name and in the path URL
- Length of the hostname and that of the entire URL
- Number of dots in the URL

The length of the hostname and that of the entire URL as well as the number of dots in the URL are proposed in [28] and have been evaluated in [27]. They are common ways to numerically characterize URLs.

Having both statistical and lexical features ready, we then encode them into a common feature space and leverage widely applied supervised learning method such as logistic regression to construct a classifier that is able to

differentiate between functional and nonfunctional sensitive transmissions.

5.3 Clustering

In addition to traffic classification, an alternative approach is to derive the malicious patterns only on collected non-functional sensitive flows. By clustering the unexpected flows into groups, we are able to generate the network-level signatures that summarize the common patterns in these flows. The signatures can then be easily fed to the existing popular NIDS platforms such as Snort⁹. We conduct an initial study on the feasibility of flow clustering in detecting privacy leakage, by deriving token-subsequence signatures [30] from the nonfunctional traffic flows found by AppInspector. Signature merging is then applied to further improve the scalability. We also conduct an evaluation in Section 6.2 to show that the derived signatures would maintain a low false positive rate.

6 EXPERIMENTAL EVALUATION

In this section, we comprehensively evaluate the effectiveness of FlowIntent, including the performance of both AppInspector and TrafficAnalyzer.

6.1 AppInspector

We have crawled more than 20,000 authentic Android apps from Google Play and Baidu App Market. Due to the different recommendation policies used in separate app markets, the apps were also crawled in distinct ways. While the top 500 apps in each category were collected from Google play, the apps from the Baidu’s app market were gathered more randomly. We also collect around 10,000 malicious samples from Drebin dataset [3] and VirusShare.¹⁰ Since we focus on PII leakages through the Internet, we only keep those apps that require both sensitive resource access and network related permissions.

For each app, we store its name and description (shown in the introduction web page, if exists), into a text file. As discussed in Section 4.1, we then run COSMOS to invoke the sensitive API calls inside the apps and store the transmissions identified by TaintDroid with tcpdump. During the

9. <https://www.snort.org/>

10. <https://virusshare.com/>

TABLE 4
App Context Classification Results and Voting
With 10 Fold Cross-Validation

Algorithm	Precision	Recall	F-measure
Random Forest	86.887%	93.283%	89.972%
Linear SVM	87.865%	93.770%	90.722%
Logistic Regression	87.353%	95.106%	91.065%
Voting	94.229%	98.852%	96.485%

running period, the UIs associated with the sensitive transmissions are also recorded into the screenshots and the hierarchy XML files. We then follow the procedure mentioned in Section 4.2 that leverages the app name, description, and screenshot to label each transmission. We have manually labeled 1,666 app contexts in total, including 651 legitimate contexts, and 1,015 illegitimate contexts, and use 10 fold cross-validation on these instances. We randomly partition all instances into 10 equal sized subsets. One of them is then chosen as the testing data, and the rest 9 subsets are used as training data. The procedure is repeated 10 times with different testing data sets.

Given TP = number of true positives, FN = number of false negatives, FP = number of false positives and TN = number of true negatives, the prediction efficiency of the model is measured by the following metrics: Precision = $\frac{TP}{TP+FP}$, Recall = $\frac{TP}{TP+FN}$, F-measure = $\frac{2TP}{2TP+FP+FN}$.

The first three rows in Table 4 give the classification results for each of the three classifiers. We note that the three supervised learning algorithms perform closely regarding our dataset. The fourth row gives the final result after voting is applied. Among the 1,666 contexts, three classifiers reach a consensus on 1,558 instances, including 947 illegal samples and 611 legal samples. Within the 1,558 instances, our model predicts 1,489 of them correctly, giving a prediction rate of 95.6 percent. The percentage of abandoned samples is low (6.7 percent for illegal cases and 6.1 percent for legal cases) and the impact is acceptable.

6.2 TrafficAnalyzer

We collected 2,125 nonfunctional flows from the 947 illegal app contexts identified by AppInspector. Among them, 116 flows are false positives as AppInspector misclassified their associated app context in the last stage. We have also collected 3,870 normal flows including both non-sensitive flows from apps that do not ask for cared permissions and functional sharing flows recognized behind the legal contexts. We then apply logistic regression with L1 or L2 regularization on these traffic flows to construct our flow classification model.

The results of our model with five-fold cross-validation are given in Tables 5 and 6. Table 5 shows the prediction result when the instances are labeled using the predictor built in last

TABLE 6
Traffic Classification Results With True Instance Label

Features	Precision	Recall	F-measure
Statistical	70.103%	85.242%	76.919%
Lexical	95.106%	96.872%	95.970%
Both	94.906%	97.726%	96.289%

subsection, from the evaluation of AppInspector, while Table 6 gives the result when true instance labels are used in the training of the traffic model. We make the following observations:

- The traffic classification model achieves 94.1 percent F-measure by using both statistical and lexical features, even when some app contexts are potentially mislabeled due to the inaccuracy of the model created by AppInspector. When the true running instance classes are used for the mislabeled 116 nonfunctional flows, the F-measure increases to 96.2 percent. Therefore, the utilization of AppInspector only incurs a slight loss in accuracy, while saving the effort of manually labeling a large number of instances.
- Lexical features alone can provide relatively good predication accuracy, which can be further improved by including statistical features. Among the set of most useful lexical features shown in Table 5, 'locat', 'jpg', 'lat', 'lng' are useful in distinguishing location and non-location flows, while the rest can be used to identify nonfunctional traffic. In particular, 'domob' is a good indicator of nonfunctional flows because it is one of the largest analytical service providers in China.
- Indicated by the F-measure results of the model using statistical features, our approach can potentially be extended to detect HTTPS flows as well, even though the lexical features cannot be applied to HTTPS traffic. The set of statistical features with the highest impact as shown in Table 5 is consistent with our observation in Section 5.2.
- We found that more than one thousand nonfunctional flows contain more than one type of sensitive resource. This is because ad and analytic services tend to collect available private data altogether and send them in a single flow. Although this observation does not give much help in predicting leakages at the network level as we do not know what data the unseen traffic will leak, it provides better app audit to detect nonfunctional flows with the help of existing taint analysis. By encoding this phenomenon into a feature, our model achieves 96.465 percent F-measure, with 93.176 percent precision and nearly 100 percent recall, using the labels assigned by AppInspector.

TABLE 5
Traffic Classification Results With the Labels Predicted by AppInspector

Features	Precision	Recall	F-measure	Attributes with highest influences
Statistical	69.100%	86.606%	76.856%	downlink packet size: mean, max, std. devn, interval between packets: mean, TCP packet count
Lexical	89.573%	97.175%	93.215%	'loc', 'stat', 'jpg', 'ads', 'domob', 'lat', 'lng', 'share'
Both	90.277%	98.281%	94.099%	

TABLE 7
Prediction Results on Non-Sensitive
Flows Reported by TaintDroid

Total reported	True nonfunctional sensitive flows	Unknown
234	185	21

Improved Sensitive Flow Detection. In addition to achieving a promising detection accuracy, our learning model is able to detect new sensitive flows that are undetectable by TaintDroid, which highlights the advantage of having network level signatures. To confirm this, we randomly select 5,510 flows generated by 670 running instances that TaintDroid does not report any leakage. Among these flows, our model detected 234 of them to be nonfunctional sharing. We then check the ground truth for each of these flows manually. The result is shown in Table 7. In the table, the ‘Unknown’ column indicates the cases where we cannot identify the ground truth, when the traffic is encrypted and the URLs are not familiar. Our model is able to detect 185 nonfunctional flows correctly, all of which are missed by TaintDroid. The result shows that our learning model is able to identify location sharing flows that are missed by host-based taint analysis, which strongly indicates the benefit of considering network level features.

Stealthy Nonfunctional Flows. As mentioned in Section 3, nonfunctional sensitive flows may hide themselves behind legal app contexts, which may introduce more damages due to the fact that users would grant the apps permission to access the protected resources. In order to detect this kind of stealthy malicious traffic, we apply the traffic classifier (trained using the labels assigned by AppInspector) to distinguish functional and nonfunctional sensitive flows with legal app contexts. 495 testing sensitive flows are picked in random. The prediction outcomes are shown in the Table 8. It reports 157 flows in total, and successfully find 142 true nonfunctional flows, giving 92.4 percent accuracy. Moreover, we can leverage the fact that ads tend to collect all sorts of sensitive data in one flow to generate a more effective model. Overall, in addition to the existing access controls, our approach offers another dimension of protection of user privacy.

Flow Clustering. Purely based on nonfunctional sensitive HTTP flows, we can also derive the clusters with similar characteristics, and generate the corresponding network signatures for well-known NIDS such as Snort. To this end, we divide the flows into groups based on the shared invariant content in their URLs and follow the procedure described in [30] to generate a token-subsequence signature for each group. Each signature represents an ordered list of invariant tokens and can be written as a regular expression. An example of the generated signatures is shown in Fig. 7, which can be written as a regular expression of the form

TABLE 8
Prediction Results on Sensitive Flows
Behind Legitimate App Contexts

Total sensitive flows	Predicted nonfunctional flows	False positives
495	157	15

$\frac{t1}{/api/analytics/v1/user[_]_android[_]_json[_]?} \frac{t2}{latlon=} \cdot \frac{t3}{\&deviceId=} \cdot \frac{t4}{\&v=} \cdot \frac{t5}{\&isLAT=} \cdot *$

Fig. 7. Example of token-subsequence signature.

$t1 \cdot t2 \cdot \dots \cdot tn \cdot *$, where each t_i is an invariant token that is identical in all the requests within the same group. Every signature is composed by the token sets of *path* and *query*. Path represents the visiting destination and it is first part of a URL. Queries are the parameters attached to URL path. In our example, t_1 is a token presenting the path of the underlying URLs and $t_2 \sim t_5$ are the queries. The regular expression can then be conveniently converted to a signature for traditional NIDS platforms that are unable to deploy machine learning models. From 2,125 nonfunctional sensitive HTTP flows, we obtain 291 network signatures in total. 15 of them (e.g., $/ \cdot * [_] _ json$) are too generic to be used alone in practice. The nonfunctional flows that contribute to these signatures convey the private data deeply in their payload (e.g., a json file) rather than directly encoding them into the URLs. Attaching their domain names to their signatures is one way to detect those flows at NIDS.

Each token-subsequence signature is a determined characteristic of specific flows, and therefore, may not scale well to cover the future variants. In order to detect as many variants as possible, we make a further refinement by merging the signatures that share sufficient similar content. We define a measure of distance between two signatures s_i and s_j as

$$d_s(s_i, s_j) = w_p d_p(s_i, s_j) + w_q d_q(s_i, s_j), \quad (1)$$

where $d_p(s_i, s_j)$ is the distance between the paths of two signatures and w_p is its pre-defined weight; similarly, $d_q(s_i, s_j)$ and w_q are the distance and the weight for the queries, respectively. $d_q(s_i, s_j)$ is calculated from the Jaccard distance¹¹ between the query token sets of two signatures. For path distances we partition the paths into sub tokens, which naturally form the sets to help calculate the Jaccard distance. $d_p(s_i, s_j)$ is therefore defined as the Jaccard distance between the path token sets of s_i and s_j . A two-dimensional distance matrix can then be computed by calculating d_s for every $i, j \in N$ where N is the index set representing all the signatures before merging. We finally feed the pair-wise distance matrix to the *single-linkage hierarchical clustering algorithm* [29], which returns a tree-like data structure whose leaves represent the original signatures and edge lengths are the distances between clusters. For instance, Fig. 8 gives an example of a hierarchical clustering tree branch.

From the branch we know that the distance between the two signatures is small so that they can be combined into a single new signature. Specifically, the 2 paths are compressed into $/api/ \cdot * [_] _ json [_] ?$ and the sets of queries can be summarized as the queries of the first signature, with $\&isLat$ removed. Putting two parts together, we get a new signature $/api/ \cdot * [_] _ json [_] ? latlon = \cdot \&deviceId = \cdot \&v = \cdot *$. We observe that certain signatures are short (e.g, length ≤ 3). In this case, we keep them untouched even if the distances among them is small to

11. The Jaccard distance between two sets A and B is $1 - \frac{|A \cap B|}{|A \cup B|}$

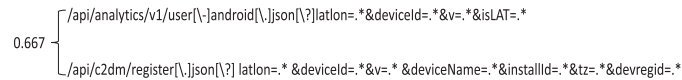


Fig. 8. An example hierarchical clustering tree branch.

avoid generating useless generic signatures. We derive 211 signatures in total after conducting the signature clustering.

The detection would be meaningless without maintaining a low false positive rate. We therefore test our 211 signatures on the 3,870 normal flows collected before. 8 of them (Table 9) generate 84 false positives (2.17 percent). After further inspection, we notice that most of them are also generic signatures, but are harder to be noticed. For example, the signature `/portal/home[.]do[?]l=.*` generates the false alarm for a legal request collected from an airline app (`http://m.staralliance.com/portal/home.do?l=en_US`). Other false positives are the HTTP requests that match the signatures generated from the leaking adwares. However, although they are truly advertisement flows, they do not convey any sensitive private data. Including these signatures is problematic. As mentioned earlier, for those signatures that may cause false positives, taking advantage of them in a conservative way, such as combining them with the hostnames, is recommended.

7 RELATED WORK

Program Analysis. Static and dynamic program taint analysis of apps focus on identifying whether sensitive data leaves the user device. For example, TaintDroid [14] tries to resolve privacy breach by modeling the behavior of an app through dynamic analysis, whereas FlowDroid [4] and DroidSafe [18] adopt static approaches to model the app behavior through byte code inspection. In addition to taint analysis, Agrigento [9] performs black-box differential analysis to detect privacy leakage. After all, all of the above approaches treat all sensitive transmissions as illegal, thus suffering from high false positive rates. BayesDroid [45] attempts to mitigate the issue by probabilistic reasoning a leak based on the proximity between the data at both source and sink points.

Detection Based on Visual Context. AppFence [21] first unveils the potential of UI data in detecting unintended nonfunctional app behavior with a customized permission control. AppIntent [54] further detects the mismatch between visual information and app behavior through symbolic execution. However, these studies requires heavy user involvement to manually inspect each exposed behavior. Whyper [33] and CHABADA [19] model app context through app's description and check it with permissions and APIs.

TABLE 9
The Signatures That Cause False Positives

<code>/portal/home[.]do[?]l=.*</code>
<code>/ExibeXML[.]php[?]USUARIO=.*&CODCIDADE=.*</code>
<code>/wxengine/rest/sst/getDates</code>
<code>/v1/query.json[?]databases=.*</code>
<code>/api/status/.*</code>
<code>/ad/getAdList.do?t=.*</code>
<code>/index[.]php[?]m=.*&token=.*&appVersion=.*</code>
<code>/api/groups[?]session[_]key=.*</code>

We argue that app description is coarse grained and does not reflect detailed run-time behaviors. AsDroid [22] defines certain policies to examine mismatch between keywords on limited visible buttons and underlying codes. Ringer *et al.* [39] design a set of GUI widgets to regulate resource access triggered from UI. They do not provide automatic solutions and cannot scale well to countless app implementations. Rubin *et al.* [40] check whether a given communication have visual impact. It treats any sensible disclosure, including the one abounding with ads, as legal, and therefore, are inevitable to ignore a lot of real harmful exposures. DroidJust [8] and LeakDoctor [48] correlate privacy disclosures with user sensible phone states to justify their legitimacy. However, the transmissions used by advertisements also cause noticeable state changes, and without further examination of the semantic of contextual information like `FlowIntent`, it would be mistakenly treated as functional. Recently, several research efforts attempt to infer context-aware security policies from users' behavioral traits [31], [49], [50]. They notice that app visibility is most influential towards the users' decisions on access control of mobile apps. Inspired by this, COSMOS [17], Backstage [5] and FlowCog [32] detect suspicious resource accesses by analyzing the textual information shown on the foreground windows with supervised learning or unsupervised learning. Their approaches only involve UI-related elements, which touch upon a subset of the features that `FlowIntent` considers. Also, without further learning from the traffic of nonfunctional accesses, they are unable to recognize malicious sensitive traffic stealthily hide themselves behind legal UI.

Traffic Profiling. Another thread of research that is relevant to our work is internet traffic monitoring and traffic classification, which has been used for determining different protocols and applications being used by the users. As discussed in [24] (and the references therein), these methods can also be used for anomaly detection [23], location categorization [13], as well as malware detection. A number of research works have focused on the detection of malicious traffic from network data (more specifically HTTP traces and URLs) using machine learning techniques [27], [35], [37]. In contrast, we focus on automatically identifying privacy disclosure caused by both authentic apps and mobile malwares with the help of context-aware modeling, instead of identifying malicious traffic generated from certain malicious samples. Recon [38] is closest to `FlowIntent`. Similar to `FlowIntent`, it also focuses on HTTP(S) flows collected from mobile devices and attempts to identify the traffic patterns leading to PII leaks. Unlike `FlowIntent` that can automatically generate suspicious traffic by exercising and inspecting the new apps, manually labeled flows are continuously required by Recon to keep pace with the fast evolution of malicious apps. More importantly, `FlowIntent` goes beyond the goal of Recon. `FlowIntent` does not limit itself to identifying PII flows only, it is able to further distinguish functional and nonfunctional flows.

TABLE 10
The Functionalities Supported by the Sensitive Transmissions

Resource	Functionality
Contact	contacts matching in social app, contact backup, etc
Storage	album, document synchronization, documents editing, etc
Camera	photo/video taking, barcode scan, etc
Microphone	voice assistant, voice message, etc

8 DISCUSSION

The Impact of Subcomponents on Accuracy. The overall performance of FlowIntent depends on the effectiveness of AppInspector and TrafficAnalyzer. AppInspector relies on two building blocks, namely TaintDroid and COSMOS. The former is a popular tool used in many important works in this field including BayesDroid and Agri-gento, while the latter is leveraged to trigger potential app sensitive behaviors. False negatives in TaintDroid and COSMOS lead to the missing of nonfunctional sensitive flows that we care about, which is unfortunately difficult to verify due to the lack of source code of the third-party applications. False positives in TaintDroid may finally result in false positives in FlowIntent such that non-sensitive flows may be misclassified as nonfunctional sensitive flows. Fortunately, one major advantage of dynamic analysis over static analysis is the much smaller number of false positives. As stated in the original paper of TaintDroid [14], on the 107 connections collected from commercial apps and reported by TaintDroid, the manual inspection of each network packet trace confirmed that there were no false positives. False positives in COSMOS result in unnecessary exploits of non-harmful program paths, but would not directly impact the overall accuracy.

For TrafficAnalyzer, when the app context and traffic reported by COSMOS-guided TaintDroid is fed to the training process, the experts could mistakenly label the instances and machine learning algorithms also have inherent accuracy limitations. A voting-based mechanism is adopted to reduce the number of false positives and false negatives at this stage. The techniques behind TrafficAnalyzer have been proven effective in various traffic-based applications and scenarios. Specifically, lexical features are well-recognized by their successful application in identifying malicious Web sites through their URLs, the resulting classifiers obtain 95~99 percent accuracy with only modest false positives [27]. ReCon [38] is another highly related work and entirely relies on the keywords in URLs and achieved 98.1 percent accuracy in identifying privacy leaks in mobile network traffic. Statistical features were used in [37] to detect malicious traffic in wireless/mobile networks and achieved 89.2 percent F-score. If necessary, payload-based inspection can be further integrated with additional cost to increase the robustness.

Generalization to Other Types of Resources. The ultimate goal of FlowIntent is to provide a generic approach to detect unauthentic leakages from all kinds of resources depending on the use cases and the distribution of real-life data. In this paper, we focus on location and phone id since they are the most exploited resources based on the malicious traffic we collected. But our approach could be extended to other kinds of resources, such as camera, microphone, storage, and contacts.

We have investigated the major functionalities fulfilled by transferring data generated by these resources and listed them in Table 10. We notice that none of them are completely transparent to users and all of them involve certain context information, which suggests that the motivation behind FlowIntent is still valid. It is possible that more advanced techniques to process app descriptions and UI data are needed to handle them, which would serve as an interesting future research topic.

Robustness. Ideally, we would like to keep the detection scheme as a blackbox for adversaries. But similar to existing learning based detection methods [3], [19], [36], [47], [53], [55], FlowIntent could be bypassed with feature engineering through carefully designed evasion logic. However, we believe the design philosophy of FlowIntent makes such attacks more difficult to succeed. Thanks to the examination of app context, the adversary can only target apps that seem to be legitimate to transfer the data of protected resources, which greatly limits the potential attack surface.

Compared to the features such as method names used in previous approaches [47], our method is more robust due to the fact that the visual data are resilient to code obfuscation. However, an adversary could come up with new obfuscation methods over visual data to bypass our detection method in the future. We envision that the integration of FlowIntent and other orthogonal work will be needed for a more advanced defense.

FlowIntent currently treats all sensitive flows under illegal contexts as nonfunctional. It is possible that a smart malware first sends sensitive data to a totally legal inquiry service provider, and then forward the feedback to a malicious receiver. This could confuse the classifier by blurring the boundary between functional and nonfunctional flows. Fortunately, there are only limited sensitive-data related lookup services and we can manually add them into a white list. It is desired to consider this kind of indirect leakage in the future.

As one of the first work in this thread, AppInspector currently does not perform further inspection on the follow-up dialogues. Fortunately, for a benign app with complex functionalities, the initial window typically conveys enough information to indicate the purpose. As indicated by Google, complication in UI design affects user experience in a negative way and can make flows, paths, and choices hard to understand [1]. AppInspector attempts to locate the Activity that leads to the sensitive access, which normally is the initial window of a specific functionality. We leave the support of multiple window inspection to future study.

Also, due to the limited time and resources, mislabelling of training data is inevitable. Thus, it is important to have more participants to reduce label noise.

Leakage Over Encrypted Channels. For encrypted channels such as HTTPS, statistical features are still helpful to detect nonfunctional traffic to some degree. Moreover, it is possible to route traffic over a controlled environment, such as a trusted virtual private network (VPN) tunnel (on-device [25], [42] or not [38]) with SSLsplit¹² integrated, to decrypt the communications over SSL and extract more complete network level features.

12. <https://www.roe.ch/SSLsplit>

Leakage Over Other Protocols. Although HTTP(S) is the prevalent channel to deliver sensitive data, it is not the only transmitting protocol. For example, some existing Android malware families such as AndroRAT¹³ directly utilize transport layer protocols (i.e., TCP and UDP) for command and control (C&C) purpose, which is transparent to the detection strategies designed for application layer protocols like HTTP(S). Extracting features and signatures for every possible privacy-disclosure protocol is helpful to mitigate the issue. In the meantime, improving the efficiency and the effectiveness of app audit to purge these malwares in advance.

Security-Centered UI Exerciser. Our prior work [15] lists a bunch of research efforts that try to automatically trigger security or privacy related behaviors in mobile apps. However, the problem of accurately preserving the user interfaces while effectively exposing sensitive behaviors largely remains open. Based on the result in [43], the coverage of the state-of-the-art GUI testing framework designed for Android is around 60 percent on 93 open-source apps. Creating a smarter security-centered UI exerciser is part of our future research.

Labelling. Manually labelling instances is time-consuming and tedious. As a starting point, “dirty work” is inevitable and we hope our effort could contribute to the entire community by making the malicious dataset we collected public.

Based on the current access control mechanism enforced on iOS and Android, billions of end users have to decide whether to grant permissions or not on their smart devices based on their understanding of app context and behavior. In other words, a great number of highly-relevant manual labelling are happening every day. Unfortunately, these labelling efforts are mostly discarded and wasted. We hope one day an industrial giant like Google or Apple can take more efforts to leverage this kind of valuable data without sacrificing user privacy. We believe the idea behind FlowIntent is a plausible way.

Exceptional Scenarios. There are exceptional scenarios that FlowIntent may not handle correctly. In particular, as FlowIntent currently focuses on textual information, it is unable to process icons that do not have meaningful texts/metadata. Image classification will be required to infer the images’ topics. Also, due to the limitation of UI exerciser, FlowIntent will have difficulty unveiling sensitive flows that may only be triggered by complicated UI interactions. Moreover, FlowIntent is built on Chinese and English samples. Extension to other languages is left for future work.

9 CONCLUSION

In this paper, we develop FlowIntent, a proof-of-concept system that makes the first attempt to automatically identify the nonfunctional privacy-leaking traffic flows from the mismatch between app context and network behavior. Compared to system level detection approaches, our network level signatures are easier to deploy at Intrusion Detection Systems to monitor a large number of devices simultaneously, while introducing zero overhead at the end hosts. FlowIntent also captures the sensitive transmissions missed by traditional taint analysis systems. In contrast to previous network level detection techniques that

rely on a given set of malicious domain names, FlowIntent can better adapt to the fast growth of app market and new leakage patterns through automated feeding of the suspicious flows generated from illegal app contexts. We have built our learning models using 2,125 privacy-sharing instances and our approach achieves about 94 percent accuracy in differentiating between functional and nonfunctional transmissions.

ACKNOWLEDGMENTS

This work was supported in part by the U.S. Army Research Laboratory Cyber Security Collaborative Research Alliance under Contract Number W911NF-13-2-0045. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation hereon. The work of Zhu was supported by the US National Science Foundation CNS-1618684. Hao Fu and Pengfei Hu are co-first authors.

REFERENCES

- [1] Google designer david hogue: How to avoid over-complication in product design. [Online]. Available: <https://modus.medium.com/google-ux-designer-david-hogue-shares-how-to-reverse-over-complication-in-product-design-and-how-90c00bcad5d7>
- [2] Legal – android app permissions. [Online]. Available: <https://www.uber.com/legal/other/android-permissions/>
- [3] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, “DREBIN: Effective and explainable detection of android malware in your pocket,” in *Proc. ISOC Netw. Distrib. Syst. Secur. Symp.*, 2014.
- [4] S. Arzt *et al.*, “FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps,” *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.
- [5] V. Avdiienko, K. Kuznetsov, I. Rommelfanger, A. Rau, A. Gorla, and A. Zeller, “Detecting behavior anomalies in graphical user interfaces,” in *Proc. IEEE/ACM Int. Conf. Softw. Eng.*, 2017, pp. 201–203.
- [6] M. Backes, S. Bugiel, and E. Derr, “Reliable third-party library detection in android and its security applications,” in *Proc. ACM Conf. Comput. Commun. Secur.*, 2016, pp. 356–367.
- [7] C. E. Brodley and M. A. Friedl, “Identifying mislabeled training data,” *J. Artif. Intell. Res.*, vol. 11, pp. 131–167, 1999.
- [8] X. Chen and S. Zhu, “DroidJust: Automated functionality-aware privacy leakage analysis for android applications,” in *Proc. 8th ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2015, Art. no. 5.
- [9] A. Continella *et al.*, “Obfuscation-resilient privacy leak detection for mobile apps through differential analysis,” in *Proc. ISOC Netw. Distrib. Syst. Secur. Symp.*, 2017.
- [10] M. Crawford, T. M. Khoshgoftaar, J. D. Prusa, A. N. Richter, and H. Al Najada, “Survey of review spam detection using machine learning techniques,” *J. Big Data*, vol. 2, no. 1, 2015, Art. no. 23.
- [11] J. Crussell, R. Stevens, and H. Chen, “MAdFraud: Investigating ad fraud in android applications,” in *Proc. ACM Conf. Mobile Syst. Appl. Serv.*, 2014, pp. 123–134.
- [12] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song, “NetworkProfiler: Towards automatic fingerprinting of android apps,” in *Proc. IEEE Int. Conf. Comput. Commun.*, 2013, pp. 809–817.
- [13] A. K. Das, P. H. Pathak, C.-N. Chuah, and P. Mohapatra, “Contextual localization through network traffic analysis,” in *Proc. IEEE Int. Conf. Comput. Commun.*, 2014, pp. 925–933.
- [14] W. Enck *et al.*, “TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones,” *ACM Trans. Comput. Syst.*, vol. 32, no. 2, 2014, Art. no. 5.
- [15] H. Fu, Z. Zheng, S. Bose, M. Bishop, and P. Mohapatra, “LeakSemantic: Identifying abnormal sensitive network transmissions in mobile applications,” in *Proc. IEEE Int. Conf. Comput. Commun.*, 2017, pp. 1–9.

13. <https://github.com/DesignativeDave/androrat>

- [16] H. Fu, Z. Zheng, A. K. Das, P. H. Pathak, P. Hu, and P. Mohapatra, "FlowIntent: Detecting privacy leakage from user intention to network traffic mapping," in *Proc. IEEE Int. Conf. Sens. Commun. Netw.*, 2016, pp. 1–9.
- [17] H. Fu, Z. Zheng, S. Zhu, and P. Mohapatra, "Keeping context in mind: Automating mobile app access control with user interface inspection," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2019, pp. 2089–2097.
- [18] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe," in *Proc. ISOC Netw. Distrib. Syst. Secur. Symp.*, 2015.
- [19] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.*, 2014, pp. 1025–1035.
- [20] R. Holley, "How good can it get? Analysing and improving OCR accuracy in large scale historic newspaper digitisation programs," *D-Lib Mag.*, vol. 15, no. 3/4, 2009.
- [21] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2011, pp. 639–652.
- [22] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "AsDroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.*, 2014, pp. 1036–1046.
- [23] O. Ibdunmoye, A.-R. Rezaie, and E. Elmroth, "Adaptive anomaly detection in performance metric streams," *IEEE Trans. Netw. Service Manag.*, vol. 15, no. 1, pp. 217–231, Mar. 2018.
- [24] H. Kim, K. Claffy, M. Fomenkov, D. Barman, M. Faloutsos, and K. Lee, "Internet traffic classification demystified: Myths, caveats, and the best practices," in *Proc. ACM Int. Conf. Emerg. Netw. Experiments Technol.*, 2008, Art. no. 11.
- [25] A. Le, J. Varmarken, S. Langhoff, A. Shuba, M. Gjoka, and A. Markopoulou, "AntMonitor: A system for monitoring from mobile devices," in *Proc. ACM SIGCOMM Workshop Crowdsourcing Crowdsharing Big Data*, 2015, pp. 15–20.
- [26] B. Liu, B. Liu, H. Jin, and R. Govindan, "Efficient privilege de-escalation for ad libraries in mobile apps," in *Proc. ACM Conf. Mobile Syst. Appl. Serv.*, 2015, pp. 89–103.
- [27] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond blacklists: Learning to detect malicious web sites from suspicious URLs," in *Proc. SIGKDD Conf. Knowl. Discov. Data Mining*, 2009, pp. 1245–1254.
- [28] D. K. McGrath and M. Gupta, "Behind phishing: An examination of phisher modi operandi," in *Proc. USENIX Workshop Large-Scale Exploits Emergent Threats*, 2008, Art. no. 4.
- [29] F. Murtagh, "A survey of recent advances in hierarchical clustering algorithms," *Comput. J.*, vol. 26, no. 4, pp. 354–359, 1983.
- [30] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically generating signatures for polymorphic worms," in *Proc. IEEE Symp. Secur. Privacy*, 2005, pp. 226–241.
- [31] K. Olejnik, I. I. Dacosta Petrocelli, J. C. Soares Machado, K. Huguenin, M. E. Khan, and J.-P. Hubaux, "SmarPer: Context-aware and automatic runtime-permissions for mobile devices," in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 1058–1076.
- [32] X. Pan *et al.*, "FlowCog: Context-aware semantics extraction and analysis of information flow leaks in android apps," in *Proc. USENIX Secur. Symp.*, 2018, pp. 1669–1685.
- [33] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards automating risk assessment of mobile applications," in *Proc. USENIX Secur. Symp.*, 2013, pp. 527–542.
- [34] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "AdDroid: Privilege separation for applications and advertisers in android," in *Proc. ACM ASIA Conf. Comput. Commun. Secur.*, 2012, pp. 71–72.
- [35] R. Perdisci, W. Lee, and N. Feamster, "Behavioral clustering of HTTP-based malware and signature generation using malicious network traces," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2010.
- [36] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "AutoCog: Measuring the description-to-permission fidelity in android applications," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2014, pp. 1354–1365.
- [37] A. Raghuramu, H. Zang, and C.-N. Chuah, "Uncovering the footprints of malicious traffic in cellular data networks," in *Proc. Int. Conf. Passive Active Meas.*, 2015, pp. 70–82.
- [38] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes, "ReCon: Revealing and controlling PII leaks in mobile network traffic," in *Proc. ACM Conf. Mobile Syst. Appl. Serv.*, 2016, pp. 361–374.
- [39] T. Ringer, D. Grossman, and F. Roesner, "AUDACIOUS: User-driven access control with unmodified operating systems," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2016, pp. 204–216.
- [40] J. Rubin, M. I. Gordon, N. Nguyen, and M. Rinard, "Covert communication in mobile applications," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2015, pp. 647–657.
- [41] S. Shekhar, M. Dietz, and D. S. Wallach, "AdSplit: Separating smartphone advertising from applications," in *Proc. USENIX Secur. Symp.*, 2012, Art. no. 28.
- [42] Y. Song and U. Hengartner, "PrivacyGuard: A VPN-based platform to detect information leakage on android devices," in *Proc. ACM CCS Workshop Secur. Privacy Smartphones Mobile Devices*, 2015, pp. 15–26.
- [43] T. Su *et al.*, "Guided, stochastic model-based GUI testing of android apps," in *Proc. 11th Joint Meet. Found. Softw. Eng.*, 2017, pp. 245–256.
- [44] G. Suarez-Tangil and G. Stringhini, "Eight years of rider measurement in the android malware ecosystem: Evolution and lessons learned," 2018, *arXiv: 1801.08115*.
- [45] O. Tripp and J. Rubin, "A Bayesian approach to privacy enforcement in smartphones," in *Proc. USENIX Secur. Symp.*, 2014, pp. 175–190.
- [46] F. Wang, Y. Zhang, K. Wang, P. Liu, and W. Wang, "Stay in your cage! A sound sandbox for third-party libraries on android," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2016, pp. 458–476.
- [47] H. Wang, J. Hong, and Y. Guo, "Using text mining to infer the purpose of permission use in mobile apps," in *Proc. ACM Int. Joint Conf. Pervasive Ubiquitous Comput.*, 2015, pp. 1107–1118.
- [48] X. Wang, A. Continella, Y. Yang, Y. He, and S. Zhu, "LeakDoctor: Toward automatically diagnosing privacy leaks in mobile applications," *Proc. ACM Interactive Mobile Wearable Ubiquitous Technol.*, vol. 3, no. 1, pp. 28:1–28:25, 2019.
- [49] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov, "Android permissions remystified: A field study on contextual integrity," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 499–514.
- [50] P. Wijesekera *et al.*, "The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences," in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 1077–1093.
- [51] M. Y. Wong and D. Lie, "IntelliDroid: A targeted input generator for the dynamic analysis of android malware," in *Proc. ISOC Netw. Distrib. Syst. Secur. Symp.*, 2016.
- [52] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective real-time android application auditing," in *Proc. IEEE Symp. Secur. Privacy*, 2015, pp. 899–914.
- [53] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "AppContext: Differentiating malicious and benign mobile app behaviors using context," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.*, 2015, pp. 303–313.
- [54] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: Analyzing sensitive data transmission in android for privacy leakage detection," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2013, pp. 1043–1054.
- [55] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual API dependency graphs," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2014, pp. 1105–1116.
- [56] Y. Zhang *et al.*, "Detecting third-party libraries in android applications with high precision and recall," in *Proc. IEEE Int. Conf. Softw. Anal. Evol. Reeng.*, 2018, pp. 141–152.
- [57] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 95–109.



Hao Fu received the PhD degree in computer science from the University of California, Davis, California, in 2019. He was supervised by Dr. Prasant Mohapatra. His research interests include network and systems security, machine learning, and program analysis. He has published more than ten papers in relevant reputed conferences, including INFOCOM, SECON, CoNext, MILCOM, etc. He was also a reviewer of CCS, CNS, ACNS, MILCOM, etc.



Pengfei Hu received the PhD degree in computer science from the University of California, Davis, California. He is currently a professor with the School of Computer Science and Technology, Shandong University. Before joining Shandong University, he was a researcher with VMware xLab. His PhD supervisor is Prof. Prasant Mohapatra. His research interests include the areas of cyber security, data privacy, mobile computing. He has published more than 20 papers in reputed conferences and journals on

these topics, including the *IEEE Communications Surveys & Tutorials*, the *IEEE Transactions on Mobile Computing*, the *IEEE Transactions on Dependable and Secure Computing*, *INFOCOM*, *CoNEXT*, *SECON*, *MILCOM*, etc. He also holds five patents in the area of mobile computing. He served as reviewer for numerous journals and conferences including the *IEEE Transactions on Information Forensics and Security*, the *IEEE Journal on Selected Areas in Communications*, the *IEEE Transactions on Mobile Computing*, the *IEEE Transactions on Wireless Communications*, *SECON*, *CNS*, *WCNC*, *MILCOM*, etc.



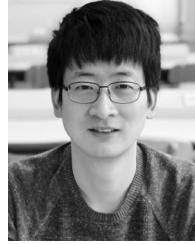
Zizhan Zheng (Member, IEEE) received the MS degree in computer science from Peking University, Beijing, China, in 2005, and the PhD degree in computer science and engineering from the Ohio State University (OSU), Columbus, Ohio, in 2010. He worked as a postdoctoral researcher with OSU from 2010 to 2014 and as an associate specialist with UC Davis from 2014 to 2016. He joined Tulane University as an assistant professor in 2016. His research interests include the areas of networking, security, and machine learning.



Aveek K. Das received the BE degree in electronics and tele-communication engineering from Jadavpur University, Kolkata, West Bengal, India, in 2012, and the PhD degree in computer science from the University of California, Davis, California, working on content-aware network data mining. He is currently a security researcher with Forescout Technologies. His current research interests include network security, data analytics, and IoT and healthcare security.



Parth H. Pathak received the PhD degree in computer science from North Carolina State University, Raleigh, North Carolina, in 2012. He is currently an assistant professor with the Computer Science Department, George Mason University. His research interests include mobile and ubiquitous computing, energy-efficient sensing, Internet-of-Things systems, wireless networking, and network analytics. He is a recipient of the Award for Excellence in postdoctoral research with the University of California, Davis, in 2015. He has also received the Best Paper Award at IFIP Networking 2014 conference.



Tianbo Gu received the BS degree from Hangzhou Dianzi University, Hangzhou, China, in 2010, and the MS degree from the University of Science and Technology of China, Hefei, China, in 2013. In 2012, he received National Scholarship which is the top award for graduate students in China. He is currently working toward the PhD degree in the Department of Computer Science, University of California, Davis, California. His research interests include Internet of Things, edge computing, and smart sensing.



Sencun Zhu received the BS degree in precision instruments from Tsinghua University, Beijing, China, in 1996, the MS degree in signal processing from the University of Science and Technology of China, Graduate School at Beijing, in 1999, and the PhD degree in information technology from George Mason University, Fairfax, Virginia, in 2004. He is currently an associate professor with Penn State University. His research interests include wireless and mobile security, network and systems security, and software security. Among

many academic services, he is the editor-in-chief of the *EAI Transactions on Security and Safety* and an associate editor of the *IEEE Transactions on Mobile Computing*.



Prasant Mohapatra (Fellow, IEEE) received the doctoral degree from Penn State University, State College, Pennsylvania, in 1993. He is serving as the vice chancellor for research with the University of California, Davis. He is also a distinguished professor with the Department of Computer Science and served as the dean and vice-provost of Graduate Studies during 2016-2018. He was the department chair of computer science during 2007-2013. In the past, he has also held visiting scientist positions with Intel

Corporation, Panasonic Technologies, Institute of Infocomm Research (I2R), Singapore, and National ICT Australia (NICTA). He received an Outstanding Engineering Alumni Award in 2008. He is also the recipient of Distinguished Alumnus Award from the National Institute of Technology, Rourkela, India. He received an Outstanding Research Faculty Award from the College of Engineering, University of California, Davis. He received the HP Labs Innovation awards in 2011, 2012, and 2013. His research interests include the areas of wireless networks, mobile communications, cybersecurity, and Internet protocols. He has published more than 350 papers in reputed conferences and journals on these topics. His research has been funded through grants from the National Science Foundation, US Department of Defense, US Army Research Labs, Intel Corporation, Siemens, Panasonic Technologies, Hewlett Packard, Raytheon, and EMC Corporation. He is a fellow of the AAAS.

▷ **For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.**