INSPIRED: Intention-based Privacy-preserving Permission Model

Hao Fu*, Zizhan Zheng†, Sencun Zhu†, Prasant Mohapatra*
*Department of Computer Science, University of California, Davis, USA.
†Department of Computer Science, Tulane University, New Orleans, USA.
‡Department of Computer Science, Pennsylvania State University, Pennsylvania, USA.

Abstract—Mobile operating systems adopt permission systems to protect system integrity and user privacy. In this work, we propose INSPIRED, an intention-aware dynamic mediation system for mobile operating systems with privacy preserving capability. When a security or privacy sensitive behavior is triggered, INSPIRED automatically infers the underlying program intention by examining its runtime environment and justifies whether to grant the relevant permission by matching with user intention. We stress on runtime contextual-integrity by answering the following three questions: who initiated the behavior, when was the sensitive action triggered and under what kind of environment was it triggered? Specifically, observing that mobile applications intensively leverage user interface (UI) to reflect the underlying application functionality, we propose a machine learning based permission model using foreground information obtained from multiple sources. To precisely capture user intention, our permission model evolves over time and it can be user-customized by continuously learning from user decisions. Moreover, by keeping and processing all user’s behavioral data inside her own device (i.e., without sharing with a third-party cloud for learning), INSPIRED is also privacy-preserving. Our evaluation shows that our model achieves both high precision and recall (95%) based on 6,560 permission requests from both benign apps and malware. Further, it is capable of capturing users’ specific privacy preferences with an acceptable median f-measure (84.7%) for 1,272 decisions from users. Finally, we show INSPIRED can be deployed on real Android devices to provide real-time protection with a low overhead.

I. INTRODUCTION

Millions of mobile applications (or apps for short) are available to users due to the fast penetration of smart devices. On the one hand, these apps access device resources to support various functionalities. For example, a weather app queries user locations to provide precise humidity information; the referral page of a utility app uses SMS to invite friends. On the other hand, they may also abuse the resources, e.g., by transmitting sensitive data to a third party that is unintended by the user or sending premium SMS stealthily to introduce extra cost to the user. To this end, mobile operating systems such as Android and iOS adopt permission systems as an important line of defense for protecting the security and privacy of users. In particular, early versions of Android present the list of permissions requested by an app when it is installed, where the users can only make an all-or-nothing decision. More recently, Android 6.0 implements an opt-in system similar to iOS, where users are allowed to grant or deny a permission to an app when it is needed by the app for the first time. But even this approach does not provide sufficient protection as an adversary can easily induce users to grant the permission first, and then exploit the same resource for malicious purposes. A recent study [56] showed that at least 80% users would have preferred to prevent at least one permission request involved in the study, and suggested the necessity of more fine-grained control of permissions. However, simply querying users for every sensitive resource access is annoying and causing dialog fatigue. Ideally, a permission system should be able to identify suspicious permission requests on the fly and automatically by taking user preferences into account, and notify users only when necessary.

To enable effective run-time permission control, it is crucial to account for the context pertinent to sensitive permission requests, as shown in several recent user studies [40, 56, 57]. They observed that the user’s preference is strongly correlated with the foreground app and the visibility of the permission requesting app (i.e., whether the app is currently visible to the user or not). The intuition is that users often rely on display to infer the purpose of a permission request, and they tend to block resource requests that are considered to be irrelevant to app’s functionalities [56]. Thus, a permission system that can properly identify and utilize foreground data may significantly improve accuracy and reduce user involvement. We posit that to fully achieve contextual integrity [14], it is important to capture more detailed foreground information beyond visibility, in order to detect the precise context surrounding a permission request.

In this paper, we propose a run-time permission system that can automatically infer user’s expectation using detailed foreground information. The main idea is to determine whether a permission request is expected by inspecting who is requesting the permission, when the request is initiated, and under what circumstances it is initiated, so that an appropriate action can be taken regarding the request (accept, deny, or notify the user). We present the design and implementation of a lightweight run-time permission control system, called INSPIRED (InteNtion-baSed PrIVacy-preSeRving pErmission moDeL). INSPIRED continuously identifies mismatches between app intentions and user intentions almost instantaneously with a low overhead. Moreover, it adapts to users’ privacy preferences on the fly, without worrying about disclos-
ing the personal decisions. In addition, INSPIRED is designed to be resilient to code obfuscation and name manipulation, encouraging adoption for monitoring commercial apps. These distinguishing features are achieved through the following key ideas.

First, INSPIRED detects unexpected permission requests through examination of contextual foreground data. As a critical part that a user interacts with, the foreground user interface (UI) of an app fulfills and reflects the underlying app functionality. For instance, a user interacting with an SMS composing page would expect the app to ask for the SEND_SMS permission once the sending button is clicked. But an SMS message sent by a flashlight instance should be considered suspicious or malicious. Even under a message composing scenario, no message should be actually sent without proper user interactions, e.g., clicking the send button.

We observe that a single widget in a window often cannot provide accurate information on app functionality. Consider the button for sending messages in an SMS app shown in Figure 1 (left). The button alone does not provide enough information on its purpose. A user needs to observe the whole page to understand its semantics. Without considering the relationships among widgets, one cannot tell if the sending behavior is legitimate or not. To this end, our approach leverages the semantic similarities at the window-level, even for windows crawled from different apps. The intuition is that benign apps tend to have a clear and informative UI to guide users so that foreground features such as words appeared on the screen reflect the underlying program logic. For instance, Figure 1 (right) shows the user interface of a different SMS app, which has similar widgets to specify message recipient, content, and the transmission behavior. Given the large number of apps with similar functionalities and UIs, it is possible to learn the correspondence between UI patterns and their semantics using foreground data crawled from popular apps, which remain valid for new apps encountered at runtime.

Second, INSPIRED adopts a two-level framework to strike a balance between usability and control. We observe that to reduce user involvement, it is important to understand app intentions for requesting a permission, so that one can tell if a permission request is necessary to fulfill app’s functionality. For instance, accessing CAMERA is necessary for scanning bar code, while requesting the SEND_SMS permission is very suspicious for a weather UI. As app intentions are independent of individual users, it is feasible and desirable to build a tool to understand app intentions automatically. However, such a one-fit-all approach is often insufficient as different users may have very different preferences on the same permission request even in a similar context [40, 57]. For instance, one user may think attaching the current location while taking a picture is appropriate, whereas another user may feel uncomfortable about potential location leakage. Previous studies suggested that predicting the decisions of one user using data collected from others has inherent limitations [40, 57]. Therefore, it is crucial to adapt the permission system by incorporating individual user’s preferences at runtime.

These observations lead us to a two-phase solution: (1) in the offline phase, we apply program analysis techniques to analyze a large corpus of sensitive apps and extract foreground data surrounding sensitive permission checks. Our approach can automatically identify the relationship between widgets. The foreground data along with the corresponding permission requests are then used to build a one-fit-all model using machine learning. (2) in the lightweight, online phase, we improve the one-fit-all model by incorporating individual users’ privacy preferences using self-adaptive learning, which can be implemented completely on the local device or assisted by remote servers (e.g., cloud-based training). Since the latter approach requires the device to transfer security and privacy related data of a user to an untrusted third party, it may introduce additional privacy concerns. Therefore, INSPIRED chooses to implement the self-adaptive learning module completely on the device so that no sensitive data would be leaked.

INSPIRED can also be combined with other runtime mediation techniques to provide protection at different levels. For instance, we can combine INSPIRED with TaintDroid [22] to offer information-flow level protection.

In summary, this paper makes the following contributions:

- We propose a novel intention-aware permission system based on app foreground information to enforce runtime contextual integrity. Our approach adopts a two-layer framework where (1) program analysis together with offline learning are used to identify UI patterns (layout and beyond) for stable app intentions that remain unchanged across users, and (2) runtime permission control and adaptive learning are used to incorporate user preferences on the fly.

- As a proof of concept, we implement a prototype of the INSPIRED permission system. INSPIRED is implemented as a standalone app and can be easily installed on Android devices with root access, without requiring OS modification. Moreover, INSPIRED is designed to be completely transparent to third-party apps. Hence, it requires no modification for apps to run under INSPIRED control. The experimental installation package can be found at [https://sites.google.com/view/inspired-mobile](https://sites.google.com/view/inspired-mobile).

- We show that INSPIRED achieves both high precision and high recall (95%) for 6,560 requests from both authentic apps and malware. Further, it is able to capture users’ specific privacy preferences with an acceptable median f-measure (84.7%) for 1,272 decisions collected from users. We further show that INSPIRED can be deployed on real Android devices to provide real-time protection with a low overhead.

II. Problem Statement

In this paper, we target threats from third-party apps who may improperly access device resources. Such threats come from either intended malicious logic embedded in an app or vulnerable components of an app that can be exploited by the attackers. We assume that the underlying operating system is trustworthy and uncompromised. We assume that apps are
isolated from each other through sandboxing and their system calls can be intervened by the permission system.

Our ultimate goal is to design a run-time permission system that enforces contextual integrity with minimum user involvement.

**Contextual Integrity:** The current permission systems of popular mobile operation systems defy user expectations over half the time since they do not consider the varying contexts of the requests [56]. We envision that to enforce contextual integrity in mobile platforms, one need to ask the following three questions regarding a permission request:

- **Who initiated the request?** An app may request the same permission for different purposes. For instance, a map app may request user’s locations for updating the map as well as for advertisement. Although it can be difficult to know the exact purpose of a permission request, it is important and feasible to distinguish the different purposes by tracing the sources of permission requests as we show in this work.

- **When did it happen?** Ideally, a permission should be requested only when it is needed. This implies that the temporal pattern of permission requests is an important piece of contextual data. For instance, it is helpful to know if a permission is requested at the beginning or at the termination of the current app activity, and if it is triggered by proper user interactions, such as clicking, long clicking, checking, etc.

- **What kind of environment?** A proper understanding of the overall theme or scenario when a permission is requested is critical for proper permission control. For instance, it is expected that different scenarios such as entertainment, navigation, or message composing may request very different permissions. In contrast to who and when that focus on detailed behavioral patterns, what focuses on a high level understanding of the context. They are complementary to each other.

We propose to answer the above three questions using the foreground data surrounding a permission request. Recent studies have shown the significance of foreground visibility in understanding user’s expectation [40, 57]. We propose to go one step further to build a run-time permission system that can capture and exploit more comprehensive foreground data from the above three perspectives.

**Minimum User Effort:** Recent studies on run-time permission control focus on characterizing users’ behavioral habit and attempt to mimic users’ decisions whenever possible [40, 56, 57]. Although this approach caters to individual user’s privacy preference, it also raises some concerns. First, users could be less cautious and the potential poor decisions made by users could lead to poor access control [57]. Second, many malicious resource accesses are user independent (although they may still be context dependent), which should be rejected by the run-time permission system without notifying the user. Furthermore, the permission system should automatically grant the permissions required for the core functional logic indicated by the context of the running app to reduce user intervention. Note that the core logic here is defined for the current dynamic context, which may not be a core functionality mentioned in the static description. For instance, an SMS message sent under “Invite friends” page after proper user interaction (i.e. clicking “Invite” button) is used to fulfill the core logic in the current context (i.e. friend referral), which may not be a main functionality of the app. Third, a user may be concerned with the liability of the system if it is being continuously monitored and analyzed. To improve usability and reduce incautious decisions, a user friendly permission system should involve user decisions only when necessary.

We propose INSPIRED, a new permission system that continuously captures semantically-meaningful information about app behaviors. INSPIRED enforces contextual integrity through comprehensive inspection of the foreground from three distinct perspectives. In particular, it answers the questions of “who”, “when” and “what” by examining the follow-
ing foreground elements:

- Activation widgets: INSPIRED models "who" by identifying the widget that triggers a sensitive resource request. A widget is a UI element shown on a foreground window, which is normally implemented using `android.view.View` and can be a button, a checkbox, etc. Users may install apps with harmful widgets injected, which leads to a severe consequence since the permissions granted by users to functional widgets can be abused by unintended ones. As shown in Figure 2, an advertisement widget that parasitizes on a weather app can stealthily collect users’ location information using location-related permissions granted to the app. Therefore, INSPIRED focuses on widget-level and considers the permissions requested by an improper widget as suspicious.

- Trigger events: INSPIRED discovers the set of events that lead to sensitive requests. A sensitive method call invoked without any prior visible event should be suspended. For example, sending a short message by clicking the send button in the message composing page of an SMS app should be considered legitimate, but no messages should be transmitted without actual user click. To verify the correctness of this temporal property, INSPIRED tracks back to the trigger event of a permission request.

- Windows: INSPIRED infers the overall theme of the environment through a full inspection of the windows. Consider the screenshots taken from the SMS apps (see Figure 1). The title New message together with the text Type message inside the window indicate a message composing environment. We further observe that windows from different apps often share a similar layout when fulfilling a similar functionality. To capture the structural properties of windows, INSPIRED maps the absolute positions of the elements in windows to their relative positions. Moreover, INSPIRED adopts a two-layer design to protect users from malicious logic with minimum user intervention, while catering to individual user's privacy preferences. The offline module of INSPIRED uses features collected from program analysis to train a one-fit-all model to capture the app intentions, through a proper modeling of benign and malicious permission requests. With on-device deployment, this model is improved by incorporating personal privacy preferences to capture user intentions at runtime. The unique features provided by INSPIRED can be summarized as follows:

  - Automatically grant necessary permission requests and reject improper ones with minimum user involvement.
  - When needed, notify users to improve decision accuracy.
  - Keep users’ decisions and behavioral data on local devices.

Overall, we achieve the following design goals:

- Intention-based detection: Our approach detects mismatches between app intentions and user intentions. It infers the purpose of a sensitive permission request through inspection of the foreground context. It stresses on contextual integrity by conducting analysis from three distinct perspectives. Our approach is able to meet users’ personal expectations through continuous updates of the on-device learning modules.

- Limited user involvement: Our system notifies a user only when the decision is user dependent and the current scenario is new to the user. In other cases, it automatically accepts or denies an app request based on the latest model with the user’s previous decisions incorporated.

- High scalability and adaptivity: Our approach is scalable to a large number of diverse permission requests. It is transparent to app source code and requires no additional developer efforts. Its accuracy and usability can be continuously improved with more apps available in the app stores and more user decisions incorporated.

- Obfuscation resilience: Previous research utilized namespace at the code level to build context-aware permission models [40, 53, 54]. However, commercial apps and malwares often modify their classes, methods and variable names to prevent reverse engineering, as shown in Figure 3. Malicious apps may further simulate the name space of official Android packages to evade detection. In contrast, our foreground-based design is resilient to code obfuscation and name space manipulation.

- Privacy-preserving: Our solution not only protects users from privacy threats caused by third-party apps, but also eliminates the potential privacy risk due to sharing user data with a third-party cloud by keeping and processing all user sensitive data on the devices.

### III. System Architecture

Figure 4 depicts the overall architecture of INSPIRED, which contains two main phases.

- Offline Phase: The offline phase is responsible for building a one-fit-all model that can be customized in the online phase. To build the model, we collect a large number of benign apps and malicious apps, and develop a lightweight static analysis technique to extract the set of sensitive API calls and the corresponding foreground windows. Subsequently, the windows are dynamically rendered to extract their layouts as well as the information

![Fig. 3: The code obfuscation adopted by a commercial app (left) and the name manipulation leveraged by a DroidKungfu malware (right).](image-url)
of their embedded widgets. The detail of contextual data
collection is given in Section IV-A. The system calls,
widgets and layouts are then used to extract features
to build a learning model that classifies each sensitive
API call of third-party apps as either legitimate, illegal
or user-dependent. Section IV-B2 describes this offline
classification procedure.

- Online Phase: In the online phase, the one-fit-all model
trained in the offline phase is customized as follows.
For each sensitive API call invoked by a third-party app,
our mediation system will intercept the call and leverage
the online learning model to identify its nature (initially,
the online model is the same as the offline model). The
sensitive API call is allowed if it is classified as legal, and
is blocked (optionally with a pop-up warning window)
if it is classified as illegal. Otherwise, the API call is
considered as undetermined and the user will be notified
for decision making. User’s decisions are then fed back
to the online learning model so that automatic decisions
are made for similar scenarios in the future. To better
assist user’s decisions, detailed contextual information
is provided in addition to the sensitive API call itself.
Moreover, we provide special mechanisms to handle
background requests without foreground contexts. We
will discuss the implementation of our online permission
system in Section V.

IV. OFFLINE ANALYSIS AND LEARNING

In this section, we discuss our approach for building a one-
fit-all model using program analysis and machine learning.

A. Foreground Data Extraction

INSPIRED models the context of a sensitive request using
the foreground data associated with the request. Although one
can manually interact with an app and record the foreground
data, it is infeasible to build a faithful model by analyzing
a large number of apps manually. An alternative approach is
using existing random fuzzing techniques such as Monkey
which generates random inputs in order to trigger as many
sensitive behaviors as possible. However, random fuzzing
is inefficient, as it may generate many inputs with similar
program behavior. More importantly, without prior knowledge
of app behaviors, random testing wastes time on exploiting
code paths that are irrelevant to sensitive resource accesses.

In this work, we propose a hybrid approach to collect
relevant foreground data, including the set of widgets, the
triggering events and the windows associated with sensitive
API calls. Our approach has two phases, a static analysis
phase and a dynamic rendering phase. In particular, we
adopt static program analysis attempts to accurately locate the
foreground components that would trigger a permission re-
quest. Compared with random fuzzing, our approach achieves
better coverage and eliminates redundant traces. The identified
foreground components are then rendered dynamically with
actual execution, which provides more complete and precise
information compared to a pure static approach. Pure static
analysis, as an over-approximation approach, is criticized by
generating false relationships between UI elements [16].

To illustrate our hybrid approach, we use the code in
Listing 1 as an example throughout this section. The code
presents the underlying logic of the open-source SMS app
QKSMS [5], shown on the right side of Figure 1.

1) Static Analysis: Our static analysis takes the entire app
package as input, and outputs its security or privacy sensitive
program behaviors, with the corresponding foreground compo-
nents identified. We detect sensitive behaviors by performing
analysis over constructed call graphs. The foreground com-
ponents that would trigger the sensitive behaviors are then
located through data flow analysis.

For each target app, we first identify its permission-
protected API calls through method signatures. We construct
a call graph for the given app with the help of FlowDroid [13]
and iterate over the graph to locate the target calls. The list of
permission-protected API methods is provided in PScout [23].
For instance, in QKSMS, sendTextMessage() at line 14
is marked as a sensitive API call that requests the SEND_SMS
permission.

The set of call graph entry points of the sensitive API calls
are then identified by traversing through the call graph. For in-
We notice that the over-approximation of the static analysis phase may introduce some misidentified UI elements that do not actually correlate with the indicated permission request. We manually filter the misidentified samples before building the learning model to lower the impact of false alarms as much as possible. However, we remark that it can be beneficial to keep some contextual instances that do not really request a permission and label them as illegal since they simulate more scenarios that should not use the permission.

2) Dynamic Rendering: For each target Activity such as ComposeActivity recognized by our static analysis, we then render it with actual execution to precisely extract its layout and widget information. Actual execution enables us to extract data of interest loaded at runtime. Capturing rendering information specified by source code is intractable for static rendering approaches such as SUPOR \[31\], which solely leverage app resource files to uncover the layout hierarchies and identify sensitive inputs. For instance, the title of the crafting page (“Compose”) of QKSMS, a critical piece of context while using the app, is declared in the Java code (line 25 in Listing 1) instead of in the resource files. Losing this kind of dynamically generated information may hinder the progress of our upcoming task to precisely infer the purpose of the underlying program behavior. Moreover, our dynamic rendering avoids further counting the falsely recognized elements introduced by the over-approximation nature of static analysis.

Most Activities cannot be directly called by default. Hence, for each app, we automatically instrument the app configuration file manifest.xml with tag <android:exported> and then repackage it into a legal apk file. After installation of the new package, we wake up the interested Activities one by one with the adb commands provided by Android. Once an Activity is awaken, the contextual foreground app data, including the layout and widget information, is then extracted and stored into XML files with UiAutomator \[6\]. We found that some Activities cannot be correctly started by this way and we ignored them for now. If necessary, we can manually interact with those cases to extract the user interfaces we need.

B. Learning

Using the extracted foreground data, we are able to build a machine learning model to detect both user-intended and user-unintended behaviors. Given a permission request, we consider it as:

**Legitimate**: if the permission is necessary to fulfill the core functionality indicated by the corresponding foreground context. The requests in this category would be directly allowed by our runtime mediation system to eliminate unnecessary user intervention. We emphasize that the core functionality here is with respect to the running foreground context, not the app as a whole. For example, some utility apps may include a referral feature for inviting friends to try this app through SMS messages. This is typically not a core functionality of the app and the developers normally do not mention this feature on
the app description page. However, the SMS messages sent under the “Invite friends” page after proper user interactions (e.g., clicking “Invite” button) should be considered as user intended. In contrast, description-based approaches [42, 43] would unnecessarily raise alarms.

**Illegitimate:** if the permission neither serves the core functionality indicated by the foreground context, nor provides any utility gain to the user. An illegitimate request can be triggered by either malicious code snippet or false program logic. The latter can happen as developers sometimes require needless permissions due to the misunderstanding of the official development documents [23].

**User-dependent:** if the request does not confidently fall into the above two categories; that is, it is not required by the core functionality suggested by the foreground context, but the user may obtain certain utility by allowing it. Intuitively, in addition to the core functionality, the foreground context may also indicate several minor features that require sensitive permissions. Whether these additional features are desirable can be user dependent. For example, besides the CAMERA permission, a picture shooting instance may also ask unnecessary permissions such as ACCESS_LOCATION to add a geotag to photos. Although some users may be open to embed their location information into their photos which may be shared online later, those who are more sensitive to location privacy may consider this a bad practice. In this case, we treat ACCESS_LOCATION as a user-dependent request and leave the decision to individual users.

1) **Features:** Before extracting features from the collected foreground contextual data, we pre-process the crawled layouts to better retrieve their structural properties. Mobile devices have various resolutions. With absolute positions, the solution derived from one device may not scale to another device. Therefore, we divide a window into nine grids and map the absolute positions to the relative positions. As shown in Figure 7, the advertisement widget is mapped to the bottom three grids, while the main frame of the app occupies the central grids.

The processed layouts are then used to extract features. As we discussed in Section IV-A, we construct three feature sets to enforce contextual integrity. More specifically, we derive the following features from a sensitive request:

- **Who:** The static phase of our foreground data collection described in Section IV-A allows us to identify the widgets leading to sensitive API calls. We then collect the feature values of the target widgets using the dynamically extracted layout files. In particular, the feature set of “who” includes the following attributes of the target widgets:
  - text: The text shown on the widget.
  - class: The Java class of the widget instance.
  - position: Its relative position in the layout.
  - size: The percentage of screen area occupied by the widget.
  - isPassword: Whether the widget is a password.
  - isClickable, isLongClickable, isCheckable, isScrollable: Whether the widget can be clicked, long clicked, checked and scrolled.

- **What:** The text shown on target widgets could be too generic (such as “Ok” and “Yes”) to convey any meaningful context. Therefore, we also derive features from the windows to help infer the overall theme of the requesting environment. We iterate over the view hierarchy of the window layout and extract all the related widgets that have text labels. For each obtained widget, we save the text displayed on the widget and its relative position in the window as features. Including both textual and structural attributes provides better scalability to capture semantical and structural similarities across millions of pages. Although developers may adopt various design styles for the same functionality, their implementations usually share a similar foreground characterization. For instance, we do not need to know whether a window is implemented with Material design. Instead, learning the title shown at the top of the window, such as “Compose” and “New message”, is crucial. These form our “what” feature set.

- **When:** The call graph traversal gives us entries of sensitive API calls. An entry point can be either a lifecycle callback or an event listener. The lifecycle of an app models the transition between states such as creation, pause, resume and termination. The event listeners of an app monitor and respond to runtime events. Both lifecycle callbacks and event listeners are prior events happened before an API call and serve as useful temporal context to the call. We therefore use the class names and method names of entry methods as the “when” feature set.

By focusing on features directly visible to users, our approach is resilient to code level obfuscation and name manipulation. Note that the entry methods are overridden of the existing official SDK APIs and cannot be renamed by the third parties.

For each of the three feature sets mentioned above, we generate a separate feature vector. Note that attributes of a widget leading to sensitive API calls appear in both the “who” feature set and the “what” feature set. However, they are treated separately to stress the triggering widget. For the “what” set, text and position from all the widgets shown on the window are included, while for the “who” set, only those related to the triggering widget are included. All textual features are pre-processed using NLP techniques before subjecting to learning algorithms. In particular, we perform identifier splitting, stop-word filtering, stemming and leverage bag-of-words model to convert them into feature vectors. The process is similar to other text-based learning methods [28, 53]. It is certainly possible to further raise the bar of potential attacks by considering more types of feature. We will discuss the feasible extensions in Section VII.

2) **Learning:** Using the three sets of features discussed above, we train a one-fit-all learning model as follows. More specifically, one classifier is trained for each permission type with a data mining tool Weka [9] using the manually labeled...
sensitive API calls related to that permission. The classifiers are trained separately for different permissions to eliminate potential interference. Each instance is labeled as either legal or illegal based on the foreground contextual data we collect, including: the entry point method signature, the screenshot of the window, and the highlighted widget invoking the API call (if there is such a widget). We ensure contextual integrity by checking whether they altogether imply the sensitive API call. The behavior is marked as illegal if it is not supported by any type of the foreground data. For instance, SEND_SMS requested under the “Compose” page without user interactions, or required by an advertisement view, is categorized as illegal.

As we mentioned in Section III, our one-fit-all models will be continuously updated at runtime to incorporate individual user’s preferences. One option is to keep sending data to a remote cloud for pruning the models. However, since the content shown on the device is often deeply personal, transmitting this kind of sensitive contextual and behavioral data out of the device would raise serious concerns on potential leaks [26]. Consider the SMS composing example again, the window may contain private information typed by the user, which is inappropriate to share with a third-party service. But the limited computational power of mobile devices makes it infeasible to repeatedly train complicated models from scratch inside the devices. To meet both the privacy and performance requirements, we apply light-weight incremental classifiers that can be updated instantaneously using new instances with a low performance overhead, which matches the memory and computing constraints of smart phones [60]. One key question is which incremental learning technique to use. To this end, we have evaluated popular incremental learning algorithms. The detailed results are given in Section VII.

V. Online Permission System

In this section, we provide the details about the implementation of our online permission system. With the help of the pre-trained model, INSPIRED automatically grants legitimate permission requests, denies illegitimate requests, and customizes the model according to user preferences.

A. Mediation and Data Extraction

To implement run-time access control, INSPIRED dynamically intercepts sensitive calls, collects features for them, and finally classifies them using an online learning model. The online model is initialized as the one-fit-all model trained offline and is customized dynamically to model user preference as discussed below.

Android does not include official APIs that allows a third-party app to mediate other apps’ requests. Instead of modifying the OS and flashing the new firmware, INSPIRED is written in Java as a standalone Android app and can be easily installed on Android devices with root access. The implementation of INSPIRED is based on XPosed [11], an open-source method hooking framework for Android. XPosed provides native support to intercept method calls, which enables us to execute our own code before and after execution of the hooked method.

To detect improper permission requests at runtime, INSPIRED dynamically extracts information from the UI elements associated with sensitive calls. Consider the example shown in Figure 5, the sendTextMessage() is triggered after clicking the mButton widget shown on the MainActivity window. INSPIRED needs to retrieve the memory references of the interested UI elements, including the running instances of mButton and MainActivity. However, simply intercepting the target sensitive call is insufficient. The problem is that although we can extract the values of the variables appeared in the current call (e.g., sendTextMessage(...)), retrieving the values from the prior calls (e.g., onClick(...) is currently infeasible in XPosed, which makes it difficult to retrieve the trigger UI instances by only hooking the sensitive API call.

To address the above problem, INSPIRED intercepts the invocations of both Activity lifecycle callbacks (e.g., performCreate(Activity) for Activity.onCreate()) and event listeners (e.g., performClickView onClick(View)) in addition to sensitive API calls. For each of these methods, it records the references of the method parameters. For instance, in the above example, the references to mButton and the Activity are stored when processing onClick(mButton). When it encounters a sensitive API call, INSPIRED retrieves the latest widget and activity it saved, and extracts the same features from them as in the offline model. In particular, “who” features are collected from the widget and “what” features are extracted from the activity by iterating over all its widgets. Moreover, INSPIRED examines call stack traces to determine the entry point methods leading to sensitive calls, which are used to derive the “when” features. Other method signatures available in the call stack can be used to build the “program namespace” features. It is possible that the latest saved widget is not the one that really triggered the sensitive request due to multi-threading. However, this rarely happens in reality and we will further discuss it in Section VII.
After converting the features into numerical values, INSPIRED uses the online learning model to predict the type of the sensitive request. It automatically grants the permission if it is classified to be legitimate with high confidence and rejects it if it is classified to be illegitimate with high confidence. For a rejected request, INSPIRED further pops up a warning to the user including the details of the request. A request that is neither legal or illegal with high confidence would be treated as user-dependent, and then creates a prompt to accept it if it is classified to be illegitimate with high confidence and rejects it if it is classified to be legitimate with high confidence. For any false automatic decision made by the system, the user can override it at the backend and perceive the cause. For any false automatic decision made by the system, the user can override it at the backend and perceive the cause.

As users can switch between Activities, a request may be initiated by a background Activity. By tracking the memory references of the associated UI elements, INSPIRED is able to reason about the background requests even if the associated UI elements are currently invisible.

B. User Preference Modeling

To incorporate user preferences, INSPIRED notifies the user if the online model identifies a request as user-dependent. Consider the example shown in Figure 6. The UI shows a product review page and a location permission is requested once the “Upload” button is clicked. On the one hand, the user may be beneficial from sharing location if the seller provides subsequent services to promote customer experience based on the user’s review and location. On the other hand, the sharing behavior could put the user at risk since there is no guarantee how exactly the location information would be used by the app developer. As the page does not provide enough evidences whether location sharing is necessary, INSPIRED treats the instance as user-dependent, and then creates a prompt to accept user decision. Our prompt not only alarms the user about the existence of the permission request, but also highlights the widget that triggered the request and the activation event.

The user decision, along with the features of the instance, is then used to update our learning model. Discussed in Section IV-B2 our classifiers are built though incremental learning in order to take care of both privacy concern and performance overhead. The incremental learning model immediately accepts the new instance and adjusts the decision strategy to better match user criteria next time.

C. Background Services

In an Android app, an Activity can start a background Service through inter-component communication. When a sensitive call is initiated by a Service, its call stack does not contain the information of the starting Activity. In this case, INSPIRED monitors the calls of Activity.startService(Intent) to track the relationship between running Activities and Services. INSPIRED can then use the information available from the Activity to infer the purpose of a Service request.

One problem with this approach is that a Service may still be alive even when the foreground Activity has finished. In this case, INSPIRED simply notifies the user about the background request and lets the user decide whether to allow or deny the request. Alternatively, we can always reject such requests. We argue that sensitive services should not exist unless they provide sufficient foreground clues to indicate their purposes. Users tend to reject requests without foreground as suggested by three recent important user studies [40, 56, 57]. Google also further restricts background services in the most recent Android O [2].

D. Defense Against GUI Spoofing

To ensure that the foreground data is indeed associated with the background request, INSPIRED ignores the widgets that are not owned by the permission requesting app. Thus, INSPIRED is resilient to GUI spoofing that tries to evade detection by hiding behind the interfaces of other apps.

More advanced GUI spoofing attacks have also been proposed in the literature [15]. For example, when a benign app running in the foreground expects a sensitive permission to be granted, a malware may replicate and replace the window of the benign app to elicit the user. An adversary may also programmatically simulate user behaviors to interact with other apps. However, such attacks can be hard to implement in practice as they require Accessibility feature [1] enabled to the malware by the user. It is worth noting that using Accessibility may play against the malware itself, since Android repeatedly warns the user about the threats caused by Accessibility. If needed, INSPIRED can also intercept the method calls initiated from Accessibility to further alarm users.

E. Handling of False Automatic Decisions

Achieving 100% precision and recall is intractable for any machine learning algorithm. To provide better usability, INSPIRED notifies the user of each rejection and provides rich contextual information, including the activation event, the triggering widget, and the screenshot, to help the user perceive the cause. For any false automatic decision made by the system, the user can override it at the backend and our incremental learning models will incorporate the user’s decision immediately.
VI. Evaluation

We evaluate the effectiveness of INSPIRED by answering the following questions:

- **RQ1:** Can INSPIRED effectively identify misbehaviors (i.e., inconsistencies between context and behavior) in mobile apps? How do the feature sets of who, when and what contribute to the effectiveness of misbehavior identification?
- **RQ2:** Can INSPIRED be applied to capture personal privacy preferences of users?
- **RQ3:** Can INSPIRED be deployed in real mobile devices with a low performance overhead?

We note that RQ1 measures the effectiveness of the one-fit-all models where individual user preferences are not involved. A request that cannot be confidently labelled as either legal or illegal is considered as user-dependent, which is not counted in RQ1. We let RQ2 capture these scenarios that rely more on user preferences. Machine learning can still help in this case using data collected from individual users.

### A. RQ1: Accuracy in Identifying Misbehaviors

We crawled more than 10,000 apps from Google Play in November 2016, all of which were top-ranked apps across 25 categories. We also used a VirusShare data set [8], which contains more than 5,000 malware samples. From these datasets, we manually labeled 6,560 identified permission requests that belong to 1,844 different apps. Each request was marked as legitimate or illegitimate through the associated foreground contextual data, including the widget (if any), the events and the window. In particular, we determined whether a request (e.g., “RECORD_AUDIO”) was initiated by an appropriate widget (e.g., a “microphone” button) after a proper interaction (e.g., clicking) and under a correct environment (e.g. voice assistant).

The sample sizes of some datasets are imbalanced. For example, the number of legal usage of CAMERA is much higher than the illegal ones. It is well known that imbalanced data can severely hinder the learning performance of classification algorithms [52]. We therefore leveraged SMOTE [18] to oversample the heavily skewed datasets before feeding them into the classifiers.

#### 1) Overall Effectiveness:

For each permission type, we leveraged the labeled requests both as training and test data in a five-fold cross validation. Specifically, we randomly divided all instances of the same permission into 5 equally sized buckets, training the classifier on 4 of the buckets, and using the remaining bucket for testing. We repeated the process 5 times and every bucket was used exactly once as the testing data. We applied cross validation on every permission type and measured the results in terms of precision, recall and F-measure [5].

As our online learning approach is a continuous training process that adapts to user decisions, a classifier that can process one example at a time is desired. To determine which machine learning technique to use, we evaluated the effectiveness of four commonly used learning methods that support incremental classification, including Hoeffding Tree, (Multinomial) Naive Bayes, Logistic Regression and (linear) SVM. Compared to non-updatable classifiers, all these methods can iteratively incorporate new user feedback to update their knowledge and do not assume the availability of a sufficiently large training set before the learning process can start [43].

A summary of the results is given in Table [I] where the mean values are calculated over all permission types. As we can see, logistic regression achieved the best result among all four classifiers. Table [III] further provides detailed results of logistic regression on each permission type. We considered 7 common permissions as for now and will investigate more permissions in the future. We observe that among all the permission types, differentiating requests of DEVICE_ID is more challenging since developers normally do not provide sufficient information in apps to indicate why the permission is requested. More human intervention could be beneficial regarding DEVICE_ID.

#### 2) Feature Comparison:

To measure how each feature set contributes to the effectiveness of behavior classification, we used the same learning technique (e.g., Logistic Regression) with different feature sets under “who”, “when” and “what” and some combinations of them, respectively. The cross validation results of RECORD_AUDIO are presented in Table [III]. Since the comparison results of other permissions share the similar trend, we omit them here.

For each feature set, we evaluated its effectiveness by comparing the accuracy of our learning models when the feature set is used and when it is not. We found that the “what” features contributed the most among the three feature sets. As we mentioned in Section [I], benign instances often share similar themes that can be inferred from window content and layout. For example, an audio recorder instance typically has a title Recorder, a timer frame 00:00 at the center and two buttons with words start and stop, respectively. From these keywords and their positions in the page, INSPIRED

### Table I: Results for Different Classifier

| Algorithm      | Median F-measure | Average Precision | Average Recall |
|----------------|------------------|-------------------|----------------|
| Hoeffding Tree | 77.9%            | 81.7%             | 78.3%          |
| Naive Bayes    | 93.9%            | 93.3%             | 92.9%          |
| SVM            | 95.5%            | 95.4%             | 95.4%          |
| Logistic Regression | 96.1% | 95.8%             | 95.5%          |

### Table II: Results for Different Permission

| Permission     | Precision | Recall | F-Measure |
|----------------|-----------|--------|-----------|
| DEVICE_ID      | 89.8%     | 89.3%  | 89.3%     |
| LOCATION       | 93.8%     | 93.9%  | 93.8%     |
| CAMERA         | 95.0%     | 95.0%  | 95.0%     |
| RECORD_AUDIO   | 96.0%     | 96.1%  | 96.1%     |
| BLUETOOTH      | 97.9%     | 97.9%  | 97.9%     |
| NFC            | 96.7%     | 96.6%  | 96.6%     |
| SEND_SMS       | 99.8%     | 99.8%  | 99.8%     |
TABLE III: Classification with Different Feature Sets

| Feature Type   | Precision | Recall  | F-Measure |
|---------------|-----------|---------|-----------|
| Who           | 81.9%     | 78.8%   | 75.7%     |
| When          | 69.7%     | 70.7%   | 70.0%     |
| What          | 95.4%     | 95.3%   | 95.3%     |
| Who & When    | 80.0%     | 79.1%   | 76.9%     |
| Who & What    | 95.6%     | 95.6%   | 95.6%     |
| When & What   | 95.6%     | 95.6%   | 95.6%     |
| Who & When & What | 96.0%   | 96.1%   | 96.1%     |

is often able to tell whether the user is under a recording theme. Although the “what” features successfully predicted most audio recorder instances, it may be of limited use in other cases where RECORD_AUDIO permission is used. For instances, developers tend to integrate voice search into their apps to better serve users. However, as the searching scenarios differ greatly from each other, it is hard to classify their intentions using “what” features only.

The “who” features help alleviate the above problem by further examining the meta data of the corresponding widget. For instance, co.uk.samsnyder.pa:id/speakButton is an image button for speech recognition, which does not provide useful “what” features as the image button does not contain any extractable textual information. However, the word “speak” in the resource-id clearly indicates the purpose of the button. In addition to the meta data, the relative position and the class attribute of a widget can help locate non-functional components, e.g., the widgets for advertisement.

We observed that for RECORD_AUDIO, the “who” features and the “when” features are highly correlated in most cases, this is because most sensitive method calls initiated by widgets are bound with the event onClick(). However, there are exceptions. For instance, com.webstar.walkies is an Internet-based walkie talkie app that transfers users’ audio information to each other. The tips “Press & Hold ” shown in its main window indicate that the recording should start only after user clicking. However, it actually starts recording once the app is open. This misbehavior can be effectively identified using the “when” features, which emphasizes that apps should request a permission only after proper user interactions.

In summary, “what” features work well in differentiating between most legitimate and illegitimate instances at the current stage. However, as malware continues to evolve, we expect that collecting more comprehensive contextual data including “who”, “when” and “what” can provide better protection. The last row in Table III shows that the combination of all the three feature sets provides the best results. Other types of features, such as the keywords extracted from hostnames, could potentially further increase the accuracy of INSPIRED. We will investigate them in the future.

B. RQ2: Effectiveness of Capturing Personal Preferences

We conducted a lab-based survey to measure the effectiveness of our models to capture individual user’s preferences, where we asked participants to classify a set of requests that were not faithfully labelled as legal or illegal in RQ1. The survey was composed and spread through Google Forms. Among the 24 participants, 3 were professors, 6 were undergraduate students and 15 were graduate students. Each user is asked to classify 50 location accessing requests collected from 40 real apps, covering several user-dependent scenarios such as shopping, photo geo-tagging, news, personal assistant and product rating. We collected 1,272 user decisions from the 24 users.

To simulate the real decision making on device, for each request, the following information is displayed to the participants: 1) Screenshot: the screenshot taken from the app right after the request was initiated, with the triggering widget highlighted. 2) Prior event: the event led to the request, such as app start and user clicking. 3) Meta-information: the app name and a Google Play link are included, whereby the participants can find more information about the app.

We evaluated the effectiveness of our user preference modeling by updating the pre-trained model constructed during the evaluation phase of RQ1 with the decisions collected from each individual user. For each user’s decisions, we randomly partitioned them into three sets and used two of the three sets as the training set to update the pre-trained model, and the rest set as the testing set. The updated model was then used to predict the decisions in the testing set. Our model yielded a median f-measure of 84.7% among the 24 users, which is reasonably good due to the limited number of samples. We expect our model to be more accurate with more user feedback.

Figure 7 presents the detailed result of each individual. A quarter of users’ results have more than 90% precision and 90% recall. Our model performed surprisingly well for one individual, with 100% precision and 100% recall. We found that some users shared very similar preferences, which leads to several small clusters. One individual tends to behave conservatively by rejecting nearly all requests, giving a sharp outlier in the lower right corner with a perfect precision but a terrible recall. We also observe that some users made
inconsistent decisions under a similar context. For instance, one user allowed a request from a product rating page but rejected another with a closely related context. The root cause of the conflicting behaviors is unclear to us, which leaves room for further improvement of our model. One possible explanation is that sometimes users are less cautious and make random decisions as suggested in [57]. Fortunately, our system can greatly help protect users from malicious behaviors caused by malware even if users make random decisions. This is because in offline training, our model has already learned many misbehaviors by malware and accordingly, it is able to block them at runtime automatically.

We also conducted a controlled experiment to test whether the finer-grained contextual info shown in our prompts can help users make better decisions. We used the screenshots similar to Figure 6 with location-based functionality at the center and a behavioral advertisement at the bottom. Without prompts, 79.2% of the participants chose to grant the permission. After being alerted that the location requests were actually initiated by advertisements, 73.9% of the users changed their minds to reject the requests. These results encourage the deployment of INSPIRED to better assist users against unintended requests.

C. RQ3: Usability on Real Devices

In this subsection, we measured the overhead incurred by INSPIRED. We installed the online module of INSPIRED on a Google Nexus 5 running Android 5.1.1 with 2.26 GHz quad-core CPU and 2GB RAM.

1) CPU Time: We installed some popular apps from different categories on the phone, interacted with them as in common daily use, and monitored the performance overhead introduced by INSPIRED. The performance data were collected using the runtime profiling tool Traceview [4], which is officially supported for debugging Android apps through tracking the performance information of each method call. We modified the device firmware to let Traceview monitor the released commercial apps without requiring their debuggable installation packages. The overhead introduced by INSPIRED is measured within the target monitored app.

Table IV shows the average CPU overhead of INSPIRED when interacting with 5 representative apps installed on the phone. Each of these apps has at least 10 million installations according to Google Play. The first column gives the average number of sensitive requests made by each app per minute. The second column shows the average total time that INSPIRED spent on inspecting a request, excluding the time waiting for user’s decisions. The third column gives the average CPU time that INSPIRED spent on a request, excluding the waiting time on I/O. The last column gives the percentage of the CPU time used by INSPIRED within an app over the total CPU time that app used during execution. Note that the value was measured within each target app, not the total CPU time used by the entire device.

We observe that INSPIRED consumed less than 5% total CPU time for all the five apps and the values vary a lot across apps. In particular, INSPIRED incurred the highest overhead on Wechat, which can be explained by two main reasons. First, Wechat intensively requests permissions when used. As a complicated communication and social app, it needs to access several sensors to provide functionalities such as voice input, location sharing, video call, etc. It also periodically reads the device ID for analytical purpose. Second, Wechat adopts its own GUI library, which takes INSPIRED longer time for analysis. Yelp and Yahoo Weather also frequently initiate sensitive requests. They continuously update locations to provide nearby services and weather information, respectively. Compared to Wechat, their UI structures are simpler and hence cost less time to analyze. Amazon asks to access microphone and location for embedded voice assistance, which has limited foreground information and was triggered only after proper user interactions. During the experiment, Paypal only initiated sensitive requests when the app was first started. The lower frequency of permission requests and the simpler UI together led to the least overhead for Amazon and Paypal.

2) Memory Usage: As the method profiling provided by TraceView did not include the memory cost, we estimated the rough memory usage of INSPIRED by dumping the runtime objects into files. We serialized the running INSPIRED objects and the related referenced objects such as Weka instances at the decision points, in which the memory use should reach the peak value. The average memory use was 5,712 KB over 50 separate decision points. Among them, over 95% memory can be attributed to the Weka machine learning module.

3) Storage: The size of the installation package of our run-time control system is 8.7 MB. After installation, the total storage occupied is 19.86 MB, including the INSPIRED classes, Xposed library, Weka library, Android support library and the resource files. We can reduce the size by discarding the unused classes files inside the libraries, and further reduction is possible by compressing some resource files.

4) Network bandwidth: INSPIRED does not generate any network traffic on its normal use. This is a significant overhead reduction compared to cloud-based systems that continuously consume bandwidth to upload user data.

VII. DISCUSSION

In this section, we discuss the limitations of our approach and make suggestions on future directions.

Features: Similar to existing detection methods based on machine learning [12, 29, 41, 43, 53, 58, 62], INSPIRED could be bypassed with feature engineering through carefully designed evasion logic. An adversary may deliberately make an app (or repackage an existing app) that contains some valid user interfaces to justify certain permission requests while piggybacking his illegitimate sensitive information flow in the same contexts. For example, he can modify an SMS app to send out user intended SMS, while at the same time, deliver messages to a malicious receiver. However, we argue that the design philosophy of INSPIRED makes such attacks more difficult to succeed. First, the adversary can only target apps that are legitimate to use the target permissions. For example,
he could only manipulate a limited number of communication or utility apps to access SMS related permissions. Second, the adversary is restricted to exploit the target permissions under proper scenes only. For example, even if he successfully elicited an end user to install the malicious SMS app, he could only send out a message under the composing page, and when to access such pages is fully controlled by the user. Thus, by enforcing contextual integrity, INSPIRED is more robust than approaches that only check description-to-permission fidelity \[29, 36, 37, 42, 43, 61\]. Third, as INSPIRED examines the trigger event and the activation widget, the adversary should carefully plug the payload into the correct position of the targeted app source code. In the example above, he cannot simply introduce a malicious background service. Instead, he should place the malicious logic inside the clicking handler of the send button to succeed. Moreover, INSPIRED can be integrated with other techniques based upon different feature sets to provide more comprehensive protection. A promising direction is to add runtime data-flow tracking support, which enables INSPIRED to better understand the semantic relationships among the widgets. In that case, an SMS is restricted to the recipient specified by the To: widget.

Although INSPIRED provides a more detailed characterization of user interface than existing approaches \[28, 33, 35\] to better detect improper permission requests, it leaves room to consider more advanced features. Moreover, an adversary who knows the precise list of features we use can potentially obfuscate the user interface to match our criteria. For example, one may put human invisible text labels (e.g., using white text on a white background) on the screen to deceive our system. Although such an attack is possible, it cannot easily bypass the current version of INSPIRED, as INSPIRED considers multiple types of UI features. As we mentioned before, our system would warn the user if it encounters confused scenarios that do not lead to a confident decision.

We envision that it is a long-term battle to fight against increasingly more advanced adversary. Our approach is flexible to incorporate more UI-related features (e.g., colors and images) to cope with emerging new attacks.

**Implementation:** As mentioned in Section \[V\] our run-time system stores the references of encountered UI elements and leverages the information available in the call stack to match sensitive API calls to the corresponding UI elements. However, the mapping could be imprecise due to multithreading. One reason is that the call stack does not contain the caller’s information of a child thread. Although we can track the initiation procedure of certain threads, there is no universal solution yet to track all possible threads inside Android apps. Even if the call stack contains the caller’s information, we may still incorrectly identify the relationship between sensitive calls and UI elements. For example, a user may click two buttons in a short time period, where only the first click leads to a sensitive call, but the time of actual invocation is later than the second click. In this case, our current implementation matches the API call to the most recently used button, which may not be the one that triggers the sensitive call. The problem could be alleviated by modifying the base code of Xposed to log the values we need inside the runtime environment.

We currently focus on the apps designed in English. However, our design could be easily extended to add multi-language support.

**Beyond Lab-based User Study:** We so far did preliminary lab-based user study in evaluating our proof-of-concept system. The demographic distribution of participants is not comprehensive and the data set is small. Once our system is ready for daily use, we will release it to popular app stores and get feedback from actual deployments beyond the controlled lab environment.

### VIII. RELATED WORK

Several previous studies have documented the limitations of mobile permission systems \[24, 34, 51, 55, 64\]. In particular, enforcing contextual integrity in mobile permission systems is considered as an important research direction. Early studies on building context-aware systems mainly depend on manually crafted policies specific to certain behaviors \[17, 19, 21, 38, 50, 63\]. More recently, researchers began to investigate methods that can automatically infer context-aware policies from users’ behavioral traits \[40, 56, 57\]. They observe that the visibility of apps is the crucial factor that contributes to users’ decisions on permission control. However, these approaches do not capture more fine-grained foreground information beyond visibility and package names.

Some recent efforts have also been made to detect unexpected app behavior from UI data. For instance, Applntnt \[59\] uses symbolic execution to extract a sequence of GUI manipulations leading to data transmissions. PERUIM \[35\] relates user interface with permission requests through program analysis. Both approaches require user efforts to locate suspicious program behaviors. AsDroid \[33\] identifies the mismatch between user interface and program behavior with heuristic rules. DroidJust \[20\] tracks the sensitive data flows to see

---

**TABLE IV**

| Target App     | Requests/min | Time/Request(ms) | CPU Time/Request(ms) | CPU Time (%) |
|----------------|--------------|------------------|----------------------|--------------|
| Wechat         | 12.6         | 174.7            | 76.7                 | 4.4%         |
| Yelp           | 5.8          | 56.4             | 21.8                 | 2.2%         |
| Yahoo Weather  | 2.5          | 42.3             | 11.3                 | 1.4%         |
| Amazon         | 0.8          | 23.0             | 8.7                  | 0.6%         |
| Paypal         | 0.4          | 27               | 11.8                 | 0.2%         |
whether they are eventually consumed by any human sensible API calls. Rubin et al. [49] detect covert communications inside mobile apps that do not trigger UI changes with control flow analysis. Roesner et al. [47] propose to regulate resource access initiated by UI elements. Ringer et al. [46] extend the idea and design a GUI library for Android. As these approaches rely on a small set of human crafted policies, they can only recognize certain misbehaviors within the domains.

Most recently, FlowIntent [28] examines all textual information shown on the foreground windows with machine learning. Though similar in spirit, it only touches upon a subset of the challenges that INSPIRED tries to address. More specifically, we extended this line of research in several ways. First, we proposed to protect contextual integrity through analyzing UI data from three distinctive perspectives: when, what and why. Second, we provided a two-layer machine learning framework that can automatically grant the necessary permission requests and reject the improper requests without requiring user involvement, as well as improving the decision accuracy based on user feedback. Third, we implemented our permission system on real devices and conducted comprehensive evaluations. Our system can be easily installed on actual devices and incurs limited overhead.

In addition to UA centric approaches, many different approaches have been proposed to detect unexpected behaviors targeting mobile platforms. Examples include WHYPER [42], CHABADA [29] and AutoCog [43], which assess description-to-permission fidelity; DroidSift [62], AppContext [58] and HSOMINER [41], which identify malwares by training on conditional API calls; SUPOR [31], UIPicker [39] and BidText [52], which detect sensitive leakage from user input; LeakSemantic [27] and Recon [45], which performs privacy protection at network layer. Moreover, Wang et.al [53] attempt to infer the mapping from permission to app functionality using class, method and variable names. Many other studies have been done to combat UI deception and spoiling [13, 25, 31, 41, 43]. These works are orthogonal to our work and can be combined with INSPIRED to further protect users.

IX. Conclusion

We propose INSPIRED, an intention-aware privacy-preserving permission system for Android. INSPIRED automatically infers the underlying program intention by examining its runtime foreground and justifies whether to grant the relevant permission by matching with user intention. It can be user-customized by continuously learning from user decisions to precisely capture user intention. It is also privacy-preserving by keeping and processing all user’s behavioral data inside her own device (i.e., without sending to a third-party cloud for training or learning).

Experiments show that our model achieves both high precision and high recall (95%) based on 6,560 requests from both benign apps and malware. Further, it is capable of capturing users’ specific privacy preferences with an acceptable median f-measure (84.7%) for 1,272 decisions collected from 24 users. Finally, we show that INSPIRED can be deployed on real Android devices to provide real-time protection with a low overhead.

ACKNOWLEDGMENT

Hidden for double blind.

REFERENCES

[1] Accessibility. https://developer.android.com/guide/topics/ui/accessibility/index.html.
[2] Android o behavior changes. https://developer.android.com/preview/behavior-changes.html.
[3] Precision and recall. https://en.wikipedia.org/wiki/Precision_and_recall.
[4] Profiling with traceview and dmtracedump. https://developer.android.com/studio/profile/traceview.html.
[5] Qksms. https://github.com/moezbhatti/qksms.
[6] Testing support library. https://developer.android.com/topic/libraries/testing-support-library/index.html.
[7] Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey.html.
[8] Virusshare. https://virusshare.com/.
[9] Weka. http://www.cs.waikato.ac.nz/ml/weka/.
[10] Wi-fi walkie talkie. https://play.google.com/store/apps/details?id=com.webstar.walkies.
[11] Xposed. http://repo.xposed.info/module/de.rov.android.xposed.installer.
[12] D. Arp, M. Spreitenbarth, M. Hubner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), 2014.
[13] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In PLDI, 2014.
[14] A. Barth, A. Datta, J. C. Mitchell, and H. Nissenbaum. Privacy and contextual integrity: Framework and applications. In IEEE Symposium on Security and Privacy (SP), 2006.
[15] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the app is that? deception and countermeasures in the android user interface. In IEEE Symposium on Security and Privacy (SP), 2015.
[16] N. P. Borges Jr. Data flow oriented ui testing: exploiting data flows and ui elements to test android applications. In I SST, 2017.
[17] S. Chakraborty, C. Shen, K. R. Raghavan, Y. Shoukry, M. Millar, and M. B. Srivastava. ipshield: A framework for enforcing context-aware privacy. In NSDI, 2014.
[18] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. Journal of artificial intelligence research, 16:321–357, 2002.
Contextual policy enforcement in android applications with permission event graphs. In Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), 2013.

[20] X. Chen and S. Zhu. Droidjust: automated functionality-aware privacy leakage analysis for android applications. In WiSec, 2015.

[21] M. Conti, V. T. N. Nguyen, and B. Crispo. Crepe: Context-related policy enforcement for android. In International Conference on Information Security, pages 331–345. Springer, 2010.

[22] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In OSDI, 2010.

[23] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In Proceedings of the ACM SIGSAC conference on Computer & communications security (CCS), 2011.

[24] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In SOUPS, 2012.

[25] E. Fernandes, Q. A. Chen, J. Paupore, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash. Android ui deception revisited: Attacks and defenses. In Proceedings of the 20th International Conference on Financial Cryptography and Data Security, 2016.

[26] E. Fernandes, O. Riva, and S. Nath. Appabstract: on-the-fly app content semantics with better privacy. In MobiCom, pages 361–374, 2016.

[27] H. Fu, Z. Zheng, S. Bose, M. Bishop, and P. Mohapatra. Leaksemantic: Identifying abnormal sensitive network transmissions in mobile applications. In Computer Communications (INFOCOM), IEEE Proceedings on, 2017.

[28] 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 Annual IEEE International Conference on Sensing, Communication, and Networking (SECON), 2016.

[29] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In IEEE/ACM International Conference on Software engineering (ICSE), 2014.

[30] R. Heartfield and G. Loukas. A taxonomy of attacks and a survey of defence mechanisms for semantic social engineering attacks. ACM Computing Surveys (CSUR), 48(3):37, 2016.

[31] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang. Supor: precise and scalable sensitive user input detection for android apps. In USENIX Security, 2015.

[32] J. Huang, X. Zhang, and L. Tan. Detecting sensitive data disclosure via bi-directional text correlation analysis. In Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2016.

[33] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. Adroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In IEEE/ACM International Conference on Software engineering (ICSE), 2014.

[34] P. G. Kelley, L. F. Cranor, and N. Sadeh. Privacy as part of the app decision-making process. In CHI, 2013.

[35] Y. Li, Y. Guo, and X. Chen. Peruim: understanding mobile application privacy with permission-ui mapping. In Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp), 2016.

[36] K. Lu, Z. Li, V. P. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), 2015.

[37] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman. A survey of app store analysis for software engineering. IEEE Transactions on Software Engineering, 2016.

[38] M. Miettinen, S. Heuser, W. Kronz, A.-R. Sadeghi, and N. Asokan. Conxsense: automated context classification for context-aware access control. In Proceedings of the 9th ACM symposium on Information, computer and communications security (Asia CCS), 2014.

[39] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang. Upicker: User-input privacy identification in mobile applications. In USENIX Security, 2015.

[40] 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 IEEE Symposium on Security and Privacy (SP), 2017.

[41] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin. Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps. In Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), 2017.

[42] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. Whyper: Towards automating risk assessment of mobile applications. In USENIX Security, 2013.

[43] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen. Autocog: Measuring the description-to-permission fidelity in android applications. In Proceedings of the ACM SIGSAC conference on Computer & communications security (CCS), 2014.

[44] C. Ren, P. Liu, and S. Zhu. Windowguard: Systematic protection of gui security in android. In Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), 2017.

[45] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes. Recon: Revealing and controlling pii leaks in mobile network traffic. In MobiSys, 2016.

[46] T. Ringer, D. Grossman, and F. Roesner. Audacious: User-driven access control with unmodified operating systems. In Proceedings of the ACM SIGSAC conference
[47] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Re-thinking permission granting in modern operating systems. In IEEE Symposium on Security and privacy (SP), 2012.

[48] D. A. Ross, J. Lim, R.-S. Lin, and M.-H. Yang. Incremental learning for robust visual tracking. International journal of computer vision, 77(1):125–141, 2008.

[49] J. Rubin, M. I. Gordon, N. Nguyen, and M. Rinard. Covert communication in mobile applications. In ASE, 2015.

[50] N. Sadeh, J. Hong, L. Cranor, I. Fette, P. Kelley, M. Prabaker, and J. Rao. Understanding and capturing peoples privacy policies in a mobile social networking application. Personal and Ubiquitous Computing, 13(6):401–412, 2009.

[51] R. Stevens, J. Ganz, V. Filkov, P. Devanbu, and H. Chen. Asking for (and about) permissions used by android apps. In MSR, 2013.

[52] Y. Sun, A. K. Wong, and M. S. Kamel. Classification of imbalanced data: A review. International Journal of Pattern Recognition and Artificial Intelligence, 23(04):687–719, 2009.

[53] H. Wang, J. Hong, and Y. Guo. Using text mining to infer the purpose of permission use in mobile apps. In Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp), 2015.

[54] H. Wang, Y. Li, Y. Guo, Y. Agarwal, and J. I. Hong. Understanding the purpose of permission use in mobile apps. ACM Transactions on Information Systems (TOIS), 35(4):43, 2017.

[55] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Permission evolution in the android ecosystem. In ACSAC, 2012.

[56] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android permissions remystified: A field study on contextual integrity. In USENIX Security, 2015.

[57] P. Wijesekera, A. Baokar, L. Tsai, J. Reardon, S. Egelman, D. Wagner, and K. Beznosov. The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences. In IEEE Symposium on Security and Privacy (SP), 2017.

[58] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In IEEE/ACM International Conference on Software engineering (ICSE), 2015.

[59] 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 Proceedings of the ACM SIGSAC conference on Computer & communications security (CCS), 2013.

[60] X. Yin, W. Shen, and X. Wang. Incremental clustering for human activity detection based on phone sensor data. In IEEE International Conference on Computer Supported Cooperative Work in Design (CSCWD), 2016.

[61] L. Yu, X. Luo, C. Qian, and S. Wang. Revisiting the description-to-behavior fidelity in android applications. In IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016.

[62] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In Proceedings of the ACM SIGSAC conference on Computer & communications security (CCS), 2014.

[63] Y. Zhang, M. Yang, G. Gu, and H. Chen. Rethinking permission enforcement mechanism on mobile systems. IEEE Transactions on Information Forensics and Security, 11(10):2227–2240, 2016.

[64] Y. Zhang, M. Yang, Z. Yang, G. Gu, P. Ning, and B. Zang. Permission use analysis for vetting undesirable behaviors in android apps. IEEE transactions on information forensics and security, 9(11):1828–1842, 2014.