Characterizing Privacy Risks of Mobile Apps with Sensitivity Analysis

Li Lyna Zhang*†, Chieh-Jan Mike Liang†, Zhao Lucis Li*†, Yunxin Liu†, Feng Zhao†, En-Hong Chen*
*University of Science and Technology of China †Microsoft Research

Abstract—Given the emerging concerns over app privacy-related risks, major app distribution providers (e.g., Microsoft) have been exploring approaches to help end users to make informed decision before installation. This is different from existing approaches of simply trusting users to make the right decision. We build on the direction of risk rating as the way to communicate app-specific privacy risks to end users. To this end, we propose to use sensitivity analysis to infer whether an app requests sensitive on-device resources/data that are not required for its expected functionality. Our system, Privet, addresses challenges in efficiently achieving test coverage and automated privacy risk assessment. Finally, we evaluate Privet with 1,000 Android apps released in the wild.

Index Terms—Mobile applications, Sensitivity analysis, Automated testing, Privacy

1 Introduction

MOBILE devices hold a lot of personal data, but many app makers are not completely transparent on their data collection practice. As a result, there have been broad concerns over privacy risks from unjustified access to sensitive resources (or data) on mobile devices [1], which vary from user location traces being uploaded unexpectedly [2] to health apps transmitting more data than necessary [3]. *Privacy risks* arise when an app unnecessarily requests sensitive data that do not contribute to its expected functionality. For instance, it is acceptable for maps to request for geolocations, but such requests might pose a privacy risk in the case of presidential campaign apps [4].

Interestingly, the public tends to put faith on the mobile platform to preserve privacy. One mechanism is to display warning before app installation to inform users of permissions required by an app. However, a survey [5] of 308 Android users and a laboratory study of 25 Android users found that only 17% paid attention and only 3% demonstrated full comprehension of such warnings. The mobile platform can allow users to configure per-app privacy settings, e.g., ON/OFF switches for individual resource categories. The academic community has extended this manual resource control with the flexibility of fabricating returned data [6], [7], [8]. The downside is that these defense mechanisms trust end users to properly configure privacy settings, especially that mobile users might not have visibility into the app nor the necessary background.

Another popular research direction is flow-sensitive taint analysis [9], [10], [11], which identifies all possible data flows from sensitive on-device sources to sinks in real time. However, such tools cannot determine whether a data flow is really necessary for an app to function properly, and manually inspecting lists of data flows can have a high burden. Furthermore, several efforts [12], [13] try to statically learn feature values of benign and malicious apps, but their classification precision relies on the feature set and app categories pre-defined. If the feature set does not significantly differentiate between malicious and benign apps, it can produce many false positives.

There are efforts using crowdsourcing to collect expert advices on an app's privacy risks [14], [15]. We believe that this approach can complement the industry's recent initiative of using risk rating as a way to improve the existing warning mechanism. Specifically, in addition to simply informing users what permissions an app requests, risk rating tries to convey the need with a number. However, most crowdsourcing approaches require the app to first be released in the wild, and a large group of experts who are instructed to fully explore the app.

We argue that the problem of risk rating can be reformulated as the proven *sensitivity analysis* [16], or an iterative process that substitutes alternative input parameter values to measure changes in output values. In the case of mobile apps, we want to determine if app outputs (i.e., user-perceived app functionality) change with different app inputs (i.e., privacy settings for an on-device resource/data). Such cases would suggest the app functionality has a high probability of dependence on the corresponding resource, and hence a lower risk rating. We note that this article focuses on sensitive on-device data, as full analysis on the closed app backend is typically not possible.

Privet is a cloud-based system that implements sensitivity analysis to assess an app's privacy risks. The system follows an explore-log-analysis pipeline. Privet first systematically automates the app while applying different privacy settings, e.g., rejecting or manipulating resource requests. During each test run, it logs app performance and behavioral metrics. Then, Privet performs sensitivity analysis between app inputs (i.e., privacy settings) and app outputs (i.e., user-perceived app functionality). This analysis decides whether there is a user-perceived equivalence between a particular privacy setting and no privacy restriction. Privet

- *L. L. Zhang, Z. L. Li and E.-H. Chen are with USTC*
  Email: {zlcherry, zhaolazy}@mail.ustc.edu.cn, cheneh@ustc.edu.cn
- *C.-J. M. Liang, Y. Liu and F. Zhao are with Microsoft Research*
  Email: {liang.mike, yunxin.liu, zhao}@microsoft.com

Manuscript received October 31, 2016; revised May 19, 2017.

realizes the following **design principles**.

First, in contrast to static analysis, dynamic analysis allows Privet to observe user-perceived app functionality such as visual and audible feedback. While one common dynamic analysis strategy is random UI exploration [17], the large testing space (due to the wide spectrum of privacy settings) renders it inefficient in achieving testing coverage. The problem exacerbates, as we consider the typical time budget allocated for testing each app is limited to minutes. On the other hand, a pure static analysis approach does not consider code blocks dynamically loaded from remote servers or rendered at run time. To this end, Privet runs *Targeted App Automation (TARA)*, a novel approach of leveraging code mining (i.e., static code analysis) to infer automation UI paths potentially affected by adjusting privacy settings, before exercising app automation.

Second, while many related efforts rely on human to identify privacy-violating apps from raw logs [9], [10], this approach does not scale well at the size of app stores. Instead, Privet assesses app privacy risks by applying sensitivity analysis to multi-dimensional user-perceivable features of app functionality. Specifically, if user interaction feedback does not change with the value of a sensitive data, the app functionality most likely does not depend on that data. Previous efforts have applied similar observations to other privacy problems [7]. To this end, Privet addresses challenges in comparing visual and audible feedback, and incorporates SVM-based classification model for assigning a privacy risk rating.

This article makes following **contributions**. We present a novel concept of applying sensitivity analysis to assess an app's privacy risks, by considering whether a requested sensitive resource would contribute to any user-perceivable app features. Then, we systematically implemented this concept as Privet, and we evaluated with top 1,000 Android apps (from Google Play) over 10 categories of sensitive data. We found that more than 48.7% of apps can have at least one type of resource requests blocked or manipulated, without impacting user perceptions. For example, the lack of home addresses from the address book does not prevent most social networking from recommending friends. Moreover, the system scalability is practical for real-world testing – compared to the standard practice of random automation, targeted app automation can achieve the same testing coverage, with an average of 85.3% less testing time. Finally, the automated assessment can achieve a classification accuracy of 93.4%, as evaluated against human labels.

2 Background and Motivations

This section motivates gaps that existing solutions cannot adaquately fill, and reformulates them into concerns that Privet addresses.

2.1 Automated Testing of Mobile Apps

The community has invested efforts in detecting and protecting users from privacy risks. Many efforts rely on dynamic analysis [10], [18] with the advantage of catching app execution contexts, and one popular realization is the "UI monkey". This subsection first introduces Android app UI controls, and then discusses the test space explosion problem faced by dynamic analysis.

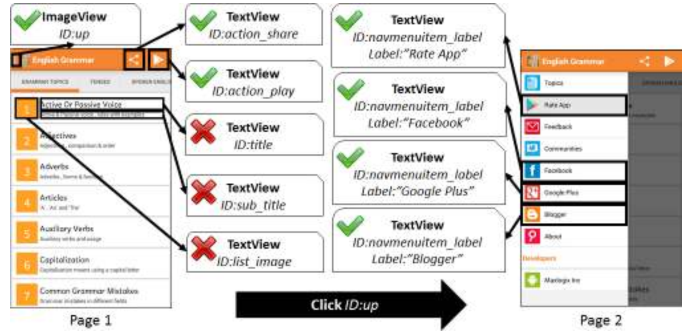


Fig. 1. To address the test space explosion, app automation should invoke only UI elements that would contribute to any sensitive data flows. In the figure, check-marked UI elements access the sensitive data sink, network. And, cross-marked UI elements do not call any sensitive system APIs.

Android App UI Elements. Generally, an Activity provides a screen (c.f. Figure 1) with which users can interact. Transitions from one Activity to another are through sending OS the Android ICC (inter-component communication) message. An Activity consists of layouts, which are built using a hierarchy of View (i.e., UI elements) and ViewGroup (i.e., layouts) objects. UI elements can be added in the following two ways. First, Android provides an XML vocabulary for developers to define the UI in an XML file. Second, developers can declare UI elements in the app code. The former is more popular, as it has less overhead in creating layouts for different screen sizes.

Not all UI elements are click-interactive (or invokable), as they are used to display texts, pictures, or even gestures. For invokable UI elements, there is typically a handler associated to handle any user invocation. For example, a Button invokes the `OnClick` event handler, and a CheckBox invokes the `OnChecked` event handler. Developers can add event handlers in either XML (e.g., `android:onClick` attribute) or run-time code (e.g., `setOnClickListener`).

***Concern #1*: Handling Test Space Explosion.** An invokable UI element can trigger its event handler to execute a sequence of system APIs and functions, which may request and consume sensitive data or resources. In our case of assessing app privacy risks, the ideal outcome is for the testing tool to invoke all UI elements that would contribute to sensitive data flows.

App UI automation can be described as an exploratory process. A black-box implementation without any knowledge of the app's inner working would simply perform random UI clicks, with the hope to achieve code coverage. However, this approach has been shown to be inefficient [19]. Fortunately, test space explosion can be addressed by leveraging one observation – Figure 1 shows two UI pages of a popular English grammar learning app. And, while there are multiple UI elements on the Page 1 (home page), some UI elements do not trigger sensitive resource requests, e.g., the three cross-marked UI elements on Page 1. Exposing such information can guide app automation and minimize any unnecessary UI exploration.

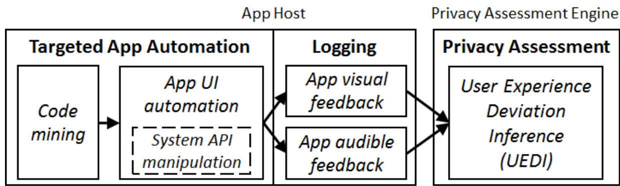


Fig. 2. System overview. (1) *Exploring* relevant parts of the app via targeted automation. (2) *Logging*. (3) *Analyzing* logs for potential privacy risks, which then help human inspectors to prioritize vetting tasks.

2.2 Quantifying and Measuring App Behavior

While time-series resource utilization (e.g., CPU and memory) is popular [17], [20], [21], they might not reveal sufficient insights for app privacy risks. For assessing app privacy risks, we argue that the user interaction feedback from the app should be the metric. This feedback can be visual (e.g., displayed contents) or audible (e.g., audio). Under sensitive analysis, if this feedback does not change with the value of a particular sensitive data, the app functionality unlikely depends on that data. Previous efforts have applied similar observations to other privacy problems [7].

***Concern #2*: Detecting App Behavioral Changes.** Detecting changes is relatively trivial for audible feedback, as inputs to audible APIs are typically file-based – changes in audio file attributes would signal changes in the auditory output. Depending on whether the file is hosted remotely or locally, attributes to compare include file name, size, date, and URL.

However, it is much more difficult to perform the same assessment for visual feedback. There are cases where two app pages look different while they can still be functionally equivalent, and an example is news apps that periodically refresh displayed contents. So, simply comparing attributes and contents of UI elements would not be sufficient. Finally, the impact of changing privacy settings might only be observable until some time later. An example is where a background service periodically downloads new contents from servers. This suggests that selecting the right app pages to examine is also important.

3 System Overview

3.1 System Architecture

Figure 2 is architectural view of the end-to-end system flow. It shows two main system components to bring sensitivity analysis to privacy risk assessment: App Hosts and Privacy Assessment Engine. Conceptually, the former automates the app and reformulates app behavior as inputs of sensitivity analysis, and the latter implements the comparison necessary for sensitivity analysis.

Technically, Privet adopts the typical explore-log-analysis app testing pipeline. Considering a social app that recommends friends with the address book on device, the exploration phase would efficiently automate this app multiple times, under a wide spectrum of privacy settings. These privacy settings dictate what data address book APIs can return. We note that app automation also allows us to accurately capture the app presentation, without dealing with language-specific analysis tools or contents loaded dynamically from remote servers. At the same time, app behavioral and performance metrics are logged. Both automation and logging are run inside *App Hosts*.

Next, app metrics logged during different test runs are analyzed and compared, by *Privacy Assessment Engine*. Intuitively, as mobile apps are mostly user-interactive, two app pages should be functionally equivalent if the user-perceivable visual and audible contents are virtually indistinguishable to users. In our previous example, if the social app displays the same list of friend recommendations regardless of the street address field, then it might unnecessarily request for too much data. Finally, based on the likelihood that an app poses privacy risks, Privet assigns a risk rating.

3.2 Challenges in Applying Sensitivity Analysis to Privacy Risk Assessment

We now discuss challenges that App Hosts and Privacy Assessment Engine need to address. These challenges arise from concerns discussed in §2.

Completeness and Efficiency of Profiling App Behavior. While app automation simplifies examining the rich interactions between apps and the environment [17], [22], it is known to be challenging to balance testing coverage and efficiency. Specifically, rich interactions explode dimensions that can be exercised. Even with only the dimension of UI interaction, there can be hundreds of UI paths to consider.

In fact, the problem exacerbates when we consider that app distribution providers tightly budget the time and resources on approving each app. This restriction limits what naive automation can catch, e.g., Google Bouncer. Therefore, the commonly employed random exploration can not well balance testing coverage and efficiency [19]. While test history can provide hints on certain automation dimensions [20], it has limitations in other dimensions such as UI, as apps (even those in the same category) can have very different UI interactions.

App Host addresses this challenge, by using static analysis techniques to learn about the app UI elements and their API usage, and then leveraging this knowledge in the dynamic context testing. In our setting, static analysis needs to learn three types of information: (1) all UI elements' attributes (e.g., ID, value, layout, activity, etc), (2) the event handler of invokable UI elements, (3) the sensitive system APIs called by the invoked event handler. Furthermore, it needs to deal with additional problems, such as multiple UI elements assigned with the same ID, a UI element being added to multiple activities, and non-standard UI elements defined by developers (Custom).

Automated Inference of App Functional Equivalence. Relying on human inspectors to look through all app logs to determine any connections between sensitive data requests and app functionality is not realistic nor scalable. Unfortunately, this is not a trivial task to automate. For instance, a news app can display contents based on the current locale and time, so straight pixel-by-pixel UI comparisons can make wrong judgment. In addition, looking at performance counters do not reveal UI information [17]. We address this challenge with Privacy Assessment Engine.

3.3 Privacy Risks Targeted by Privet

We formally define privacy risks as unnecessary requests of sensitive resource/data that do not contribute to how end-users would perceive core app functionality. As with all security and privacy tools, there are scenarios out of scope for Privet, as elaborated below.

First, many apps require backend servers for data processing, aggregation and storage. However, without the server-side source code, it is difficult for any tool to fully validate privacy compliance. A nuanced example is when the backend might claim to store only aggregated data (e.g., fitness data), but it unexpectedly shares raw data with the third-party companies. Full server-side code disclosure is necessary for validation. Second, apps can be written in ways that are difficult to automate. For example, purely gesture-based apps have no explicit UI element to discover and invoke. Since games make up a large fraction of these apps, Privet currently does not inspect games. Finally, some data sources may be sensitive depending on their contexts. An example is input text fields that can hold values from passwords to random strings. Detecting such cases can benefit from techniques (such as natural language processing) that are out of scope for Privet.

4 App Host

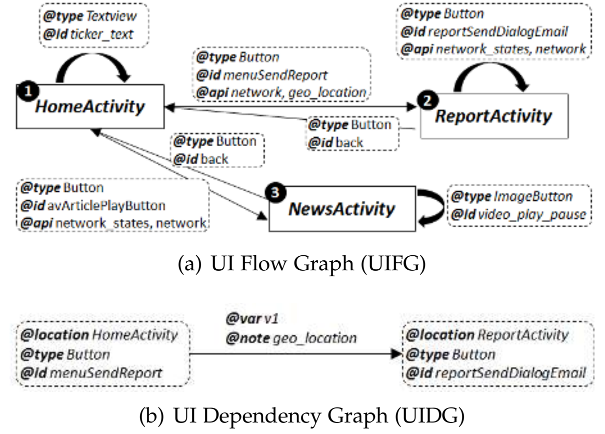
This section describes App Hosts features. Individual App Hosts are isolated environment that can be physical mobile devices or emulators on the cloud. The isolation simplifies setting up parallel and clean-slate tests.

4.1 Targeted App Automation (TARA)

App automation can be driven by various workloads. While being popular, the random automation workload has shortcomings in balancing testing coverage and efficiency metrics [19]. In contrast, Privet improves both metrics of *app automation* (i.e., dynamic analysis) with inputs from *code mining* (i.e., static code analysis). Two observations suggest this approach is practical for mobile app testing – First, not all code paths request sensitive data. Second, the majority of mobile app packages (e.g., Android Java-based apps, Windows managed code) can be decompiled.

Generating App-specific Automation Models. For Privet, app automation workload models are graph representations combining control, data, and UI flows. The widely known control flow graph (CFG) and data flow graph (DFG) do not have the necessary UI information. Instead, Privet builds the UI Flow Graph (UIFG) and the UI Dependency Graph (UIDG). Figure 3 illustrates both graphs with a news app.

In UIFG, each node represents an app page layout, and each directed edge represents a page layout transition triggered by a UI invocation. We annotate edges with (1) the triggering UI element's ID and attributes (e.g., value and type), and (2) a list of sensitive APIs called as a result of the UI invocation. The first step to construct UIFG is to discover all page layouts and UI elements – in the case of Android app packages, the included manifest lists all page layouts (i.e., activities), and layout XML files detail all UI elements (e.g., ID, name, type, and text). The second step is to map individual UI elements



(a) UI Flow Graph (UIFG)

(b) UI Dependency Graph (UIDG)

Fig. 3. Partial UIFG and UIDG of a news app. The former shows transitions among three page layouts, and the latter shows one dependency in the app code.

to the app page that they are shown on. Privet searches the decompiled app binary for code blocks related to UI initialization, and an example is `btnStart = (Button) findViewById(R.id.btnStart)`. In this case, Privet also needs to find the variable holding the resource ID of `R.id.btnStart`, and it backward-searches in the code for the last variable assignment. The final step is to analyze the code that handles UI invocation (e.g., `onClick` callback method), to look for ***(i)*** calls to sensitive system APIs, and ***(ii)*** calls to initiate any page layout transitions (e.g., `LayoutInflater.inflate` method). Also, we note that code can be indirectly triggered through inter-component communication (ICC) via Intents, and we leverage existing efforts to find such code blocks [23]. At this point, by merging all layout transitions discovered, we get the UIFG.

In UIDG, each node represents a UI element, and each directed edge indicates that the destination node has a data dependency on the source node. For example, an app can have a `SEND` button that uploads the geo-location data collected by clicking the `LOCATE` button. Technically, Privet leverages static taint analysis efforts (e.g., [24]), to determine whether a variable is shared by two UI invocation handlers.

Automating Apps. Ideally, Privet exercises only UI paths that request (and possibly depend on) some sensitive resources. And, the same UI paths are exercised several times, while manipulating resource requests under different privacy settings. Both UIFG and UIDG guide TARA to preselect such paths during run-time, as described below.

Given a targeted sensitive resource, TARA starts by searching in UIFG for edges, which represent UI element invocations (UI_{target}) that can trigger such resource requests. Then, starting from the UIFG root node (or app's home page layout), TARA tries to iteratively make progress towards UI_{target} from the current node. During each step, TARA invokes one UI element on the current page layout, and this UIFG edge is determined by running Dijkstra's algorithm over UIFG. Dijkstra's algorithm guarantees the traversed UI path to be shortest. Being iterative allows TARA to deal with any mismatching between UIFG and dynamic analysis, as explained later.

At this point, TARA searches in UIDG for UI elements

TABLE 1
Different dimensions of API data manipulations explored by Privet

	API Data Manipulations
Phone states	Device ID and phone number
Network states	Cellular and Wi-Fi status
Network	Data payload sent and received
Geo-location	Random location within 500m and 1,000m radius, and different countries
Contacts	Emails, phone numbers, names
Photo albums	Photo content
Browser	History and bookmarks
SMS messages	# of entries returned
Calendar	# of entries returned
Accounts	# of entries returned for each account type

(UI_{dep}) that depend on the sensitive data requested by UI_{target}. If so, TARA also needs to invoke these UI elements (by running Dijkstra's algorithm again). This covers the case where two code blocks (triggered by different UI elements) pass sensitive data via code variables, for instance. Finally, this statically generated end-to-end UI path is recorded for subsequent test runs under different privacy settings.

One complication is that some page layout transitions can depend on dynamic contents, or conditional variables that are not easily satisfiable. Therefore, even with UIFG and UIDG, TARA's dynamic analysis tool can land on an unexpected page layout or fail to find the expected UI element. In either case, TARA starts an iterative process to try to find another UI element that would also lead to UI_{target}, by moving backwards in the sequence of previously traversed page layouts. In the worst case, TARA resorts to random app automation to make progress. Given this complication, the recorded UI path might not be the shortest, and we describe Targeted Path Pruning (TPP) next to address this problem.

Compacting UI-automation Paths. If TARA resorts to random app automation, it runs Targeted Path Pruning (TPP) to compact UI-automation paths. We formulate the TPP problem as follows – a UI path P has nodes $\{C_1, C_2, ..., C_n\}$ representing a sequence of app page layouts, and edges represent UI elements to invoke. Assuming that C_n is our targeted node, TPP aims to skip UI element invocations that would lead to cycles in UI automation.

To illustrate TPP, we consider the following UI path of a simple news app: {*article_list*, *article_1*, *article_list*, *article_2*, *article_list*, *article_3*, ...}. Suppose our target page is *article_3*, TPP recognizes that *article_list*s are the same page, and skips the first four invocations in the path.

An important consideration is the dependency between two app pages. In our previous example of news app, it is possible that reaching *article_3* requires reaching *article_2* first. So, we use a backward-traversing approach that looks for the first available app page, starting from the end of P. In the example above, TPP first looks for *article_3* on the page of *article_list*. If it cannot find it, then TPP searches for *article_2* on the page of *article_list*. If this step succeeds, TPP retains *article_2* in the final path, in order to reach *article_3*.

4.2 Manipulating Sensitive Data Requests

Table 1 shows different sensitive APIs targeted by Privet. Depending on privacy settings of each test run, App Hosts can manipulate the arguments and return values of sensitive API calls in two ways – the first way is to empty the return value, and the second way is to replace the return value with fake data (in place of the true data). App Hosts can fake data based on the requirement of consistency (over a testing session) and deviation (with respect to the true data). For the former, an example is where App Hosts provide a phone ID that stays the same for an entire testing session, or changes with each call. The latter determines the content to return – in the case of address book requests, App Hosts can manipulate some entry fields.

The space of possible API arguments and return values introduces challenges. Fortunately, apps that are (functionally) similar should typically behave similarly to the same type of input data. For instance, while trying all possible geo-locations would be time-consuming, most news apps work well with country-level locations, and most weather apps need only city-level locations. Therefore, the functional category of an app (which is also typically used by app stores) can give hints on prioritizing experiment parameters.

5 Privacy Assessment Engine

Privacy Assessment Engine runs sensitive analysis on one app at a time. Adopting One-at-a-Time (OAT) methodology for sensitive analysis, the engine iteratively compares logged app behaviors from each privacy setting with the baseline of no intentional system API manipulations. The output is a binary answer: true (i.e., app behaviors change significantly without the corresponding sensitive data), and false (i.e., otherwise). This is different from efforts that verify app UI compliance against predefined policies [25].

One key question to answer is the comparison metrics to use for sensitivity analysis. For the privacy risks targeted by Privet, ideally, these comparison metrics should adequately tie to user-perceivable app functionality. We note that performance counters, e.g., latency of web request/reply and disk I/O, are not necessarily the defining features here. The reason is that they are either indirectly reflected on the screen, or can be masked through caching. Instead, interactions between an app and the user are typically in both audible and visual form.

Technically, Privacy Assessment Engine proposes User Experience Deviation Inference (UEDI) to compare visual and audible outputs of the same app under different privacy settings. We now formulate the problem that UEDI targets. Given logs from two runs of an app, UEDI estimates the potential magnitude of user-perceived differences, and this potential is presented as the UEDI score. The score is between 0 and 100. The higher the score, the lower the potential difference. So, for Privet, this implies the app's functionality does *not* depend on the corresponding sensitive data.

Since UEDI assigns a score to each edge in the UIFG graph, a UI-automation path would have a set of scores. This set of UEDI scores is then converted to the binary output, based on some thresholds. Since the impacts of a system API manipulation might show up only in subsequent app pages, aggregating all UEDI scores can handle this case.

5.1 Evaluating Differences in Visual Outputs

In most cases, app functionality can be visually perceived by users – for example, a news or weather app displays

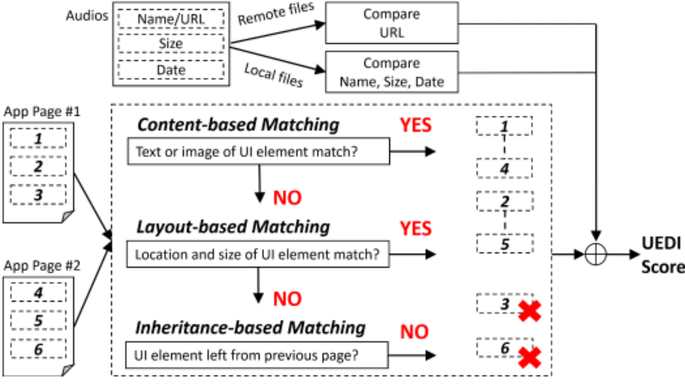


Fig. 4. UEDI evaluates the magnitude of differences between two app pages. This evaluation is based on both visual and audible outputs, where each output type follows a different pipeline.

real-time contents, and a social app displays feeds and messages. Even local apps (e.g., media players, alarm clocks, and calculators) have GUI. Therefore, we argue that metrics defining the end-user visual perception are: ***(1)*** the content values visible on the screen, and ***(2)*** the screen layout.

Interestingly, classifying visual features above can be challenging due to dynamic app contents. Simple comparisons of screen contents, e.g., pixel-by-pixel comparisons, do not work well in this case. Considering the example of weather apps, real-time content updates should not impact end-user experience. To this end, $\text{UEDI}_{\text{Visual}}$ is a set of classification techniques to automatically evaluate differences in visual outputs.

The basic idea behind $\text{UEDI}_{\text{Visual}}$ is to compare corresponding app pages of two runs. Each comparison is based on matching up as many individual UI elements as possible. $\text{UEDI}_{\text{Visual}}$ considers the following element meta-data: type, text or image (if applicable), location, and size. Intuitively, the more UI elements that can be matched up between two app pages, the more similar these two app pages should be, and hence higher score. Next, we present the steps that $\text{UEDI}_{\text{Visual}}$ implements (c.f. Figure 4).

Content-based Matching. The first step is to match up UI elements of the same type based on their content values (such as texts and images), and to count successful matches, $Num_{Content}$. We note that UI element IDs to be unreliable as some mobile platforms do not enforce developers to assign an element ID to all UI elements, e.g., list items in Android. Also, $\text{UEDI}_{\text{Visual}}$ does not yet consider the visual layout of each UI element (e.g., location and size) because many apps can reorder screen contents based on the current mobile device contexts (e.g., time and geo-location).

Fuzzy textual matching can have less false negatives than exact matching. For instance, many news apps place a real-time view count besides each news article. $\text{UEDI}_{\text{Visual}}$ computes the widely used edit distance – the total number of insertions, deletions, and substitutions necessary to turn one string into another. Then, if the ratio of this distance to string length is less than $Threshold_{Edit_Dist}$, it considers both UI elements to be the same. As §6 mentions, we use human labels to learn this threshold.

At this point, $\text{UEDI}_{\text{Visual}}$ counts $Num_{Content}$. For UI elements that cannot yet be matched, UEDI will try to match them with layout information, as described next.

Layout-based Matching. This second step relies on the UI element type, location and size. The output is the number of successful matches, Num_{Layout}. From our empirical observations, many apps use highly dynamic page contents to provide users with real-time information or the feeling of freshness. An example is the recommendation list in radio apps. In these cases, while two runs of experiments display different contents, they can still be functionally equivalent.

It is possible that some UI elements are still not matched up after this step. Interestingly, these UI elements could be residues or artifacts from previously viewed page(s), and we describe how UEDI deals with these elements next.

Inheritance-based Matching. The third step tries to count how many of the not-yet-matched UI elements are residues or artifacts of the previously viewed page(s). These UI elements should not be considered by $\text{UEDI}_{\text{Visual}}$, as they are not the results of viewing the current app page layout. For example, some apps implement page overlay (e.g., pop-ups) to display information on top of visual contents from the previous page. In these cases, UI elements not in the overlay page should be ignored. We note that this step can not be performed as the first step as many apps display identical content items on consecutive pages.

To do this, $\text{UEDI}_{\text{Visual}}$ computes the intersection of the UI element list of the current page layout and the previous one. Then, it outputs the size of this intersection list, $Num_{Residue}$.

Computing $\text{UEDI}_{\text{Visual}}$ Score. Through the multi-step matching process, each app page on a UI-automation path gets three numbers: $Num_{Content}$, Num_{Layout}, and $Num_{Residues}$. UEDI then computes the $\text{UEDI}_{\text{Visual}}$ score, $UEDI_Score_{Visual}$, for individual app page layouts, as Equation 1 illustrates:

$$UEDI_Score_{Visual} = \frac{Num_{Content} + Weight \times Num_{Layout} + Num_{Residues}}{Total\ Num\ UI\ Elements} \quad (1)$$

Since layout-based matching is a heuristic, rather than exact content matching, we use $Weight$ (between 0 and 1) to scale down its significance. Our experience with 1,000 Android apps suggests that layout-based matching has an accuracy of about 80%, so we set $Weight$ to be 0.8.

5.2 Evaluating Differences in Audible Outputs

For many app categories, audio can be a primary user interaction channel. Audio-related APIs generally accept file name as the input argument, which points to either a locally or remotely hosted audio file. Therefore, inferring user-perceivable changes can be achieved by comparing the file meta-data including name, size, and date. This approach of comparing file meta-data is similar to many off-the-shelf file synchronization tools. However, since it is not trivial to obtain meta-data for remotely hosted files, Privet compares the URL in such case.

As illustrated by Equation 2, $\text{UEDI}_{\text{Audible}}$ has a binary output. A score of 0 means the audible output has a high

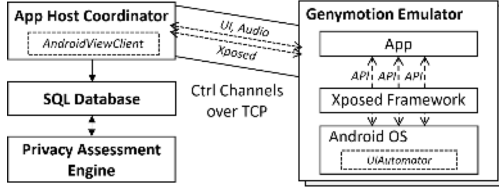


Fig. 5. Implemented components of Privet prototype.

dependency on the privacy setting being tested. And, a score of 1 suggests otherwise.

$$UEDI_Score_{audible} = 0 \; or \; 1 \tag{2}$$

5.3 Computing UEDI Score

Equation 3 shows that computing the UEDI score considers both the $\text{UEDI}_{\text{Visual}}$ score and the $\text{UEDI}_{\text{Audible}}$ score.

$$UEDI_Score = UEDI_Score_{visual} \times UEDI_Score_{audible} \tag{3}$$

We note that the computation uses multiplication to produce the same effect of logical conjunction. If the $\text{UEDI}_{\text{Audible}}$ score is 0, then the UEDI score would also be 0. On the other hand, if the $\text{UEDI}_{\text{Audible}}$ score is 1, then the $\text{UEDI}_{\text{Visual}}$ score would determine the UEDI score.

There are various strategies to aggregate UEDI scores of app pages from each test run: average, max, min, etc. Privet takes the min score, i.e., the most significant impact from a particular API manipulation on the entire run. Furthermore, since different app categories (as already defined by app stores) can exhibit different behavior and design, category-specific thresholds might be needed to convert the UEDI score. Thus, for each app category, we train category-specific thresholds from human labels with Support Vector Machine (SVM) [26], which is a very popular classification algorithm with high accuracy and robustness.

Additional Considerations. In addition to out-of-scope cases listed in §3.3, we acknowledge that a resourceful attacker can devise countermeasures to fool Privet – e.g., intentionally changing either visual or audible outputs, regardless of the reason behind requesting sensitive data. One solution is to multiply the UEDI score by a weight, to factor in the relative user expectation for individual resources. Suppose geo-location is strongly assumed to be unnecessary for calculator apps, its UEDI score can be multiplied by a weight for that app category. Privet can leverage efforts on modeling user expectations for permissions to determine the weight value for each app category [27].

6 Implementation

This section describes our implementation decisions. Figure 5 illustrates the overall integration. The entire Privet implementation consists of ∼53,012 lines of code (loc) – App Host: 45,396 loc, App Host Coordinator: 2,200 loc, Privacy Assessment Engine: 5,416 loc.

6.1 App Host

App Hosts currently support Android apps, and each instance has its own Genymotion emulator [28], which is one of the fastest emulators available on the market. In addition, Genymotion supports sensor emulations such as GPS. We describe the three main components of App Hosts next.

App Automator. The construction of UIFG and UIDG relies on Androguard [29] and Apktool [30]. Androguard includes an Android app packages (APK) decompiler and static analysis tools for instructions and permissions. Apktool decodes resources (e.g., app page layouts) in app packages. From these outputs, our Python toolchain extracts UI and control flows, by mapping caller-callee relationships of methods, and associating registered UI events and callbacks. In addition, we decode properties (e.g., `component`, `action`, `extra`, and `data`) of Android ICC Intents, and link two code blocks if there is an Intent between them.

App Host drives UI automation with the Appium [31] UI library. Appium starts the app that matches the given Android app ID, class name and text (optional). Then, it obtains the user interface (UI) view tree describing attributes of all UI elements on the current app page: class name, ID, screen coordinates, text contents, etc. App Hosts can enter texts into fields with pre-specified UI elements IDs, and this can be useful in entering login credentials. Apps can collaborate to accomplish a task, e.g., the restaurant recommendation app passes control to the map app for driving navigation. App Host stops the test once the target app is not running in the foreground. The same applies to clicking a URL that opens the web browser.

Finally, we note that App Host resets the testing environment prior to each session. This process includes clearing app caches and system states (e.g., system clock and started apps). Furthermore, for apps that require sign-in, developers are already required to provide a working credential when making app submissions to distribution providers. To enter credential, TARA can look for UI elements whose meta IDs resemble login. Otherwise, the current industry practice is to resort to manual inputs.

System API Manipulation. To intercept Android system API calls, we have implemented one Xposed [32] module for each API category in Table 1. Since most app distribution providers (e.g., Microsoft) have dedicated in-facility testbeds for app testing, rooting devices is not a roadblock. We implemented a JSON-based (over TCP) command channel to control our Xposed modules. This way, App Host Coordinator can pass arguments such as target app name, target API to manipulate, and API manipulation strategies.

Logging App Behavior. Each run of tests produces several log files: system API calls, page layouts and screenshots after each click, sensitive data blocked or injected, and a network traffic dump. Due to the emulator's limited storage, we periodically upload logs to App Host Coordinator every 10 minutes.

For producing audible outputs, Android provides two sets of APIs: `SoundPool` and `MediaPlayer`. The former can play small audio clips that are stored on the local disk, and the latter is for longer music and movies hosted locally or remotely. We note that a remotely hosted

audio file can also be pre-cached locally, which is then considered as a local file by Privet. We use the Xposed framework to capture arguments of three functions: ***(1)*** `SoundPool.load`, ***(2)*** `MediaPlayer.create`, and ***(3)*** `MediaPlayer.setDataSource`. For remotely hosted audio files, we log the URL string. For locally hosted audio files, we log the file name, size and date.

6.2 App Host Coordinator

Currently, we maintain a pool of Android-based App Hosts on a private cloud, and all hosts emulate the popular Nexus 5 device. A central server manages all App Hosts, and it schedules and sends commands to execute testing tasks. Each App Host has two JSON-based (over TCP) command channels to App Host Coordinator: one is for sending user interaction commands, and another one is for sending Xposed commands. High-level instructions are translated into commands of these two channels.

6.3 Privacy Assessment Engine

We highlight how two types of thresholds are set: $Threshold_{Edit_Dist}$ and UEDI score thresholds. First, we derive $Threshold_{Edit_Dist}$ from the evaluation dataset, which is collected from nine human users looking at 8,153 pairs of different app pages. With a $Threshold_{Edit_Dist}$ of 0.1, $\text{UEDI}_{\text{Visual}}$ can well determine whether two similar strings should be classified as being functionally identical (e.g., news title with a real-time viewer count). Second, we keep separate UEDI score thresholds for individual app categories, and we use the same categories as the ones on Google Play. In fact, from our human labels, SVM fails to identify a single threshold for all app categories, and this further motivates the design of having separate thresholds.

7 Evaluation

This section is organized by the following major results: ***(i)*** Targeted App Automation reduces our total testing time by 85.3%, as compared to popular random exploration; ***(ii)*** UEDI can determine whether two app pages are functionally equivalent with an accuracy of 93.4%, as evaluated with human labels; ***(iii)*** More than 48.7% of apps can have at least one type of resource requests blocked or manipulated.

7.1 Methodology

We downloaded 1,000 Android apps from the Google Play store for system evaluations. Instead of a random selection of apps, our set covers a diverse spectrum of popular apps. Specifically, we bucketized the top 50 apps of all 45 store categories this year into eight groups, and these groups were defined by three dimensions: text-heavy vs. image-heavy, networked vs. isolated, and sensor-dependent vs. sensor-independent. Finally, we took 125 most popular apps from each of the eight groups, for a total of 1,000 apps. Table 1 lists targeted categories of sensitive system APIs.

Experiments started by generating UI-automation paths for each app, with respect to event handlers that call sensitive system APIs. Since mobile system APIs typically do not overlap in functionality, manipulating multiple APIs at a time would not reveal additional insights. Then, we exercised each UI-automation path without any API manipulation, to obtain the app behavior baseline. Next, for each UI-automation path, we manipulated the corresponding API in one way. We logged screenshots, performance counters, audio file attributes, and UI information, as described in §6. Noticeably, we found 11.3% apps need sign-in credentials (most of them are Social and Music & Video apps), and we created accounts and logged in before testing.

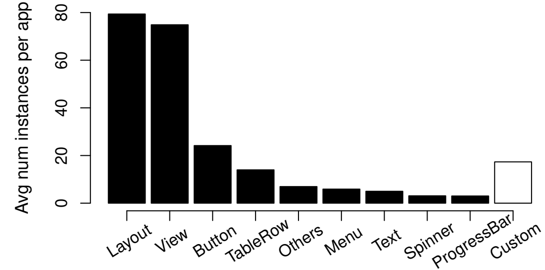


Fig. 6. While developers can include custom UI controls, most invokable UI elements are still Android UI controls – ***(i)*** Layout: `LinearLayout`, `RelativeLayout`, `FrameLayout`, `TableLayout`. ***(ii)*** View: `View`, `ImageView`, `ScrollView`. ***(iii)*** Button: `Button`, `ToggleButton`, `ImageButton`, `RadioButton`. ***(iv)***: Text: `TextView`, `EditText`.

7.2 Targeted App Automation (TARA) Component

This section quantifies how TARA reduces the overall testing time from ***(i)*** pre-selecting potentially interesting UI-automation paths with static code analysis, and then ***(ii)*** compacting these paths with TPP.

Extracting UI Elements in Real Apps. We start by showing statistics about invokable UI elements from our Android app pool. This result is important as automation tools cannot automate non-invokable UI elements. As explained previously, UI elements can be of two types: Android-base or Custom UI elements. Figure 6 shows the average number of invokable UI elements per app, for the most commonly used categories of UI elements. We note that, while apps can have Custom UI elements, Android-base UI elements still dominate. And, there are UI-related libraries readily available to interact with these UI elements.

Among invokable UI elements in an app, TARA calculates the UI-automation path to the ones that request permissions to sensitive resources. Figure 7 shows the average number of these sensitive UI elements per app. There is an average of 6.59 invokable UI elements per app accessing the Network (which is the highest number), an average of 1.56 and 0.56 invokable UI elements for geo-location and account, respectively. This observation suggests that, while Android apps can have an average of 163.21 of invokable UI elements, having insights into these elements can significantly reduce the testing time.

Pre-selecting UI-automation Paths. Given that TARA selects UI-automation paths that would trigger sensitive system API calls, the prerequisite is being able to discover and map these API calls to UI elements. This directly translates to the testing coverage achievable. Figure 8 suggests that, for more than 66% of apps, TARA can statically discover and

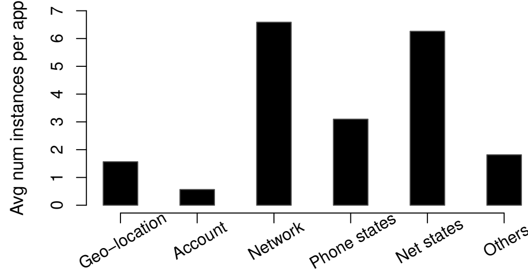


Fig. 7. Average number of UI event handlers that call sensitive system APIs.

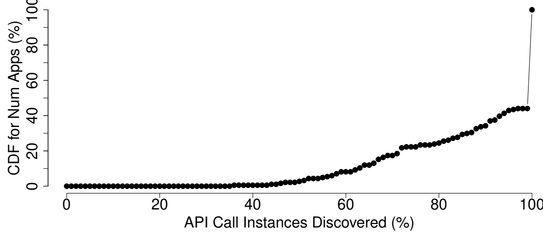


Fig. 8. CDF of apps that TARA can discover API call instances.

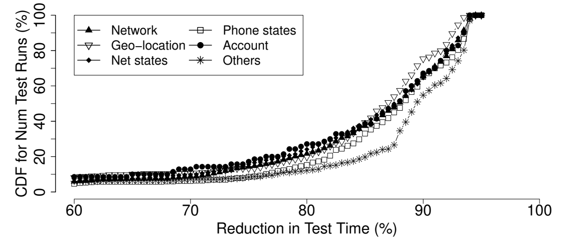


Fig. 9. CDF of test runs for which TARA can reduce the testing time.

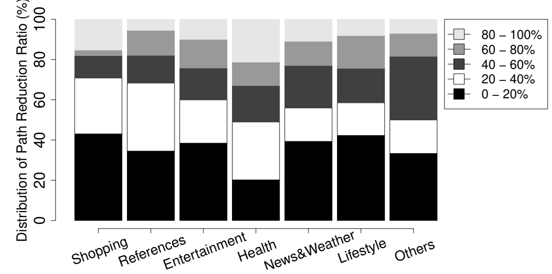


Fig. 10. Histogram of TPP results organized by app categories.

map *at least* 90% of individual calls of sensitive system APIs. We found that one major reason behind mapping failures is the dead code blocks left by developers.

Next, we illustrate the advantage of driving app automation with static code mining. Our comparison baseline is the standard industry practice of random workload. Due to the random nature, we repeat each privacy setting 100 times and report the average. Given the sensitive APIs listed in §7.1, we record the amount of time necessary for both TARA and random exploration to reach all calls in the code. Specifically, this can be measured by considering the UI elements that trigger these calls. Figure 9 shows the CDF of number of apps, with respect to the reduction in testing time achieved. On average, TARA can achieve the same testing coverage with 85.3% less testing time.

While TARA can significantly reduce the testing time, we note that analyzing code for pre-selecting UI-automation paths does not take much time. On a PC with a 2.5 GHz of quad-core CPU and 16 GB of memory, it takes about 65.8 sec to analyze one app package (with an average size of 5.2 MB).

Exercising Pre-selected UI-automation Paths. Since static analysis lacks run-time contexts, some UI-automation paths pre-selected might not be feasible to be automated. Privet logs the progress of each testing session, and cross-referencing this information with decompiled app binaries reveals the following insights.

First, some execution paths need human intervention such as taking a photo. Such cases are known to be difficult for dynamic testing techniques, and Privet resorts to random walking. In our experiments, 65.4% of the infeasible paths are fail due to this reason. Second, crafting appropriate inter-process messages can be difficult, and even current state-of-art tools can achieve only 85.0% of accuracy in the best case [23], [24]. In Android, inter-process communication is implemented by ICC (inter-component communication) that can carry data of arbitrary format. While source code analysis can provide some hints on what ICC messages should look like, this approach can handle simple cases. Third, some UI elements are on a path containing dead code, and they cannot be executed in the real world. In addition, we also found some UI elements triggered at the alarm scheduled tasks are also fail to execute at most time, due to the reason that TARA does not wait for alarms to wake up. Finally, we also observed few cases where the Appium UI library fails to detect all UI elements on an app page.

Overall, excluding paths with a length of 1, Privet can successfully exercise 81.7% of pre-selected UI-automation paths (and reach the targeted UI element).

Targeted Path Pruning (TPP). As mentioned in §4, in the worst case where TARA resorts to random app automation, TPP compacts UI-automation paths to minimize testing time. To evaluate the effectiveness of TPP, we randomly explore all apps in our repository. Specifically, for each random UI-automation path, we mark the nodes (i.e., page layouts) that must be visited, as these nodes can trigger sensitive API calls. Empirical results show that 81.3% of random UI-automation paths can be compacted, with an average length reduction of 35.7% (with a deviation of 0.28). In addition, 32.6% of the paths can be compacted by more than 50%. Therefore, this translates into a significant reduction in testing time, especially in large-scale app testing. In our case, TPP reduces the total testing time by 56.2%.

Furthermore, we note that the effective amount of path reduction varies with the app category. Figure 10 shows the distribution of path reduction ratio grouped by app category. This depends on the general app UI structure of each app category. For instance, many news apps rely on links at the home page to reach other pages. Therefore, frequent transitions from/to home page result in UI loops that can be removed by TPP.

TABLE 2
Accuracy and false rates for UEDI to decide whether two app pages are functionally equivalent.

	Accuracy	False Positives	False Negatives
Shopping	89.4%	5.7%	4.9%
References	94.2%	4.2%	1.6%
Entertainment	94.5%	3.9%	1.6%
Health	96.1%	2.8%	1.1%
News & Weather	90.4%	6.0%	3.6%
Lifestyle	92.4%	4.5%	3.1%
Others	92.0%	6.3%	1.7%

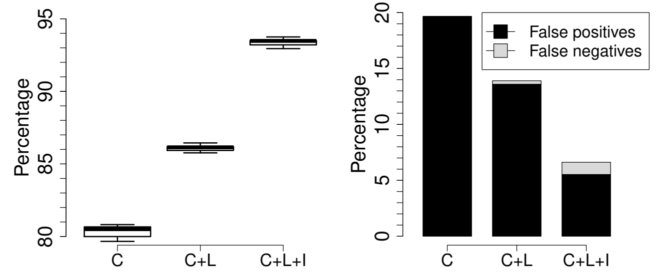
7.3 User Experience Deviation Inference (UEDI) Component

Evaluation metrics for the UEDI mechanism are the false positive rate and the false negative rate – cases where UEDI mistakenly decides two app pages are functionally equivalent and different, respectively. While false positives would result in over-conservative privacy protection (and thus user inconvenience), false negatives would result in relaxed privacy protection (and thus a false sense of protection). To provide evaluation baselines, we asked nine volunteers to label 8,153 pairs of different app pages, based on whether they think each pair is functionally equivalent. Participants range from 20 to 45 years old, and they include one female and eight males. Two are from US, and seven are from China. And, they work or study at IT companies and institutes. We note that the variance of labels among these users is reasonably small.

Table 2 shows the false positive rate and false negative rate, per app category. In most cases, Privet can decide whether two app pages are functionally equivalent with an accuracy above 90.0%. The table also suggests that Privet has false positives, or cases where it wrongfully thinks two app pages are functionally equivalent. And, since the UEDI score depends on both the $\text{UEDI}_{\text{Visual}}$ score and the $\text{UEDI}_{\text{Audible}}$ score, we next delve into both.

$\text{UEDI}_{\text{Visual}}$. 80% of data points (i.e., $\{UEDI_{Visual}$ score, human label$\}$ tuples) from each user are used by SVM to train $\text{UEDI}_{\text{Visual}}$ classifiers, and 20% are for testing. Privet achieves an accuracy of 93.4% on average, with a false negative rate of 1.1% and a false positive rate of 5.5%. Empirical data suggest that these false rates come from two dominant sources: content loading problems due to server and network performance (9.7% of false positives and 23.1% of false negatives), and user preferences (68.1% of false positives and 65.4% of false negatives). For the former, a typical case is where pages are not completely loaded as expected – i.e., some page contents are not displayed, and $\text{UEDI}_{\text{Visual}}$ is unable to make the right decision. We note that having visibility and control on the backend and content delivery network are impractical and out of scope for this work. For the latter, we have observed cases where users have different opinions on whether two app pages are functionally equivalent. In one such case, a sports app automatically changes the default display language if the current geo-location is available. And, there was a disagreement on whether this app really needs the geo-location data.

Figure 11 illustrates the gain from each of the three $\text{UEDI}_{\text{Visual}}$ matching strategies, by using labeled data from the same nine volunteers. In effect, disabled strategies would return zero matching pair. While the content-based matching step can already achieve an accuracy of 80.4%, the layout-based and the inheritance-based steps improve it to 86.1% and 93.4%, respectively.



(a) Avg positive rates (boxplot) (b) Avg false rates

Fig. 11. Each $\text{UEDI}_{\text{Visual}}$ matching strategy improves the overall positive rate, and lowers the false rates, as evaluated with human labels. (C) Content-based matching, (L) Layout-based matching, (I) Inheritance-based matching.

TABLE 3
Distribution of audio sources in Android apps.

	Remotely Hosted	Locally Hosted
News & Weather	35.8%	64.2%
Shopping	20.0%	80.0%
Lifestyle	47.9%	52.1%
Reference	33.3%	66.7%
Entertainment	36.1%	63.9%
Travel	20.0%	80.0%
Music	47.9%	52.1%
Health	40.0%	60.0%
Tools	48.0%	52.0%
Others	25.9%	74.1%

$\text{UEDI}_{\text{Audible}}$. Among the pool of 1,000 apps, we found 21% of them access audio-related APIs. This suggests that audible outputs should not be overlooked by Privet. Table 3 shows the distribution of the two audio sources: locally hosted files and remotely hosted files. We note that some apps can use both sources. Interestingly, remotely hosted files occupy a large fraction of the distribution, so wrongfully manipulating the network access can change audible outputs. In this case, the $\text{UEDI}_{\text{Audible}}$ score is equal to 0. Finally, empirical results suggest 3.08% of app pages fall under this case.

7.4 Comparisons on Privacy Risk Assessments

We compare Privet with baseline obtained from two sources: human labels and related efforts.

Comparisons with Human Baseline. To obtain human labels, we ask eight human inspectors employed by an app distribution provider to go through 3,236 instances of risk assessment, for a total of 25,888 data points. For each instance, human inspectors are presented with app descriptions, name of the sensitive resource manipulated, and two app pages under two privacy settings. Then, they decide whether that particular app would ***(i)*** legitimately need the resource, ***(ii)*** or not need it, ***(iii)*** or be uncertain from the app descriptions. On average, Privet agrees with

TABLE 4
App categories with lowest user acceptance of privacy assessment.

	Resource	Average	Std Dev
Shopping	Network	90.5%	0.054
	Geo-location	96.8%	0.031
	Net states	96.4%	0.022
	Phone states	98.9%	0.047
	Others	97.8%	0.016
References	Network	98.9%	0.011
	Geo-location	98.7%	0.022
	Net states	97.8%	0.011
	Phone states	98.0%	0.008
	Others	94.6%	0.031
News & Weather	Network	95.1%	0.018
	Geo-location	98.0%	0.025
	Net states	95.2%	0.039
	Phone states	96.2%	0.041
	Others	95.5%	0.008
Lifestyle	Network	98.3%	0.011
	Geo-location	99.2%	0.022
	Net states	95.4%	0.043
	Phone states	94.0%	0.045
	Others	98.1%	0.021

TABLE 5
Risk assessment from Privet and the related effort, PrivacyGrade.

	Agreement	Disagreement
Geo-location	89.3%	10.7%
Network	88.2%	11.8%
Phone states	82.5%	17.5%

TABLE 6
% of apps whose requests for sensitive resources might pose privacy risks. Results are organized by the resource type.

	LOW	MEDIUM	HIGH
Account	68.4%	15.8%	15.8%
Geo-location	71.9%	12.5%	15.6%
Net states	43.0%	20.3%	36.7%
Network	24.6%	22.3%	53.1%
Phone states	61.9%	21.0%	17.1%
Others	58.1%	9.3%	32.6%

TABLE 7
% of apps whose requests for *network* might pose privacy risks.

	LOW	MEDIUM	HIGH
Shopping	0.0%	50.0%	50.0%
References	33.3%	19.1%	47.6%
Entertainment	31.3%	28.1%	40.6%
Health	54.5%	9.1%	36.4%
News & Weather	10.4%	17.9%	71.7%
Lifestyle	41.2%	11.7%	47.1%
Others	31.6%	36.8%	31.6%

human labels 96.9% of time, and it can achieve this accuracy without significant human overhead. Table 4 highlights four categories with the worst acceptance rate. We note that the Shopping category has the lowest acceptance rate due to two apps that change the app background color, and different users react differently.

Comparisons with Related Efforts. PrivacyGrade [27], [33] is a publicly available database listing calculated Android app privacy risks. It performs static analysis to analyze the permission usage in top 400 third-party libraries, and then it builds models by crowdsourcing people's expectation of resource usages for each app. Grades are assigned to individual apps based on the deviation between people's expectation and what the app actually requests for. The grade is between A and D, where D suggests high potential of privacy risks. Next, we compare Privet with PrivacyGrade from two levels: individual permissions and apps.

60.2% of our app pool have also been evaluated by PrivacyGrade. PrivacyGrade tries to classify the need of three main resources (i.e., phone states, network, and geo-location) into 10 potential usages: `app functionality`, `user payment`, `utility usage`, `targeted advertising`, `social service`, `mobile analytics`, `market/customer analysis`, `content providers`, `game engines`, and `SMS SDKs`. We label `app functionality` and `utility usage` as legitimate permission usage, and remaining categories are labeled as illegitimate. We then compare the labels with Privet.

Table 5 shows results about the three resources, and it suggests that PrivacyGrade and Privet achieve an average 87.4% of agreement on assessing the need for these resources, and 12.6% of disagreement. We looked into the disagreement cases and found Privet performed better in several cases. First, we find that some apps can work and show no behavioral differences when we fake the device ID. Therefore, while Privet thinks that requests to device ID can be regulated to preserve privacy, PrivacyGrade thinks the device ID is used for some undetermined app functionality. Second, most disagreement cases are due to the fact that PrivacyGrade does not analyze an app's internal code, but only third-party libraries. So, if a privacy-violating data flow is present in the app's internal code, it would not be detected by PrivacyGrade.

To compare the privacy risks at app level, we consider apps in our pool with a PrivacyGrade grade lower than A, and see whether Privet also identifies the same privacy risks. We label an app as B if there are 25%- 50% of permissions can be manipulated or blocked, and label an app as C if the percentage is between 50% with 75%, as D if the percentage is above 75%. Results show that PrivacyGrade and Privet agree 93.6% of the time. Most instances of disagreement are due to the same piece of sensitive data being used by both app functionality and potential privacy-violating actions. In these overlapping cases, Privet would allow the data request to pass as not to impact any app functionality, but PrivacyGrade will assign a low grade to reflect those potential privacy-violating behaviors.

8 Deployment Experience and Case Studies

This section presents our deployment experience in rating app privacy risks for an app distribution provider. The pilot classifies each app into three labels according to UEDI classification results: *LOW (risk)*, *MEDIUM (risk)*, and *HIGH (risk)*. Particularly, *LOW* implies all requests are classified as functionally necessary, and *HIGH* implies none of them is necessary. Otherwise, we label it as *MEDIUM*.

Risk Rating Distribution. The first question we want to answer is the distribution of apps with respect to the three classification labels above. Table 6 shows a 28.5% highly suspicious sensitive data usage, and a 16.8% less suspicious

TABLE 8
% of apps whose requests for *geo-locations* might pose privacy risks.

	LOW	MEDIUM	HIGH
Shopping	96.2%	2.0%	1.8%
References	97.0%	2.0%	1.0%
Entertainment	83.3%	0.0%	16.7%
Health	50.0%	50.0%	0.0%
News & Weather	65.8%	18.4%	15.8%
Lifestyle	66.7%	1.3%	32.0%
Others	82.3%	1.0%	16.7%

TABLE 9
% of apps whose requests for *phone states* might pose privacy risks.

	LOW	MEDIUM	HIGH
Shopping	61.5%	23.1%	15.4%
References	53.8%	30.8%	15.4%
Entertainment	66.0%	26.4%	7.6%
Health	76.9%	1.4%	21.7%
News & Weather	59.3%	18.7%	22.0%
Lifestyle	61.5%	15.4%	23.1%
Others	63.0%	25.9%	11.1%

behavior ("MEDIUM"). Then, Table 7, 8, and 9 illustrate the three most frequently accessed sensitive resources on mobile devices – network, geo-location and phone state. Some modern apps depend on network to deliver user-perceived content, and they are susceptible to network-related API manipulations. Likewise, since News and Weather apps typically use geo-location to provide localized content, they are relatively sensitive to location-related API manipulation. Finally, phone states are typically used to maintain browsing history, but substituting them with fake IDs might impact the visual contents perceived by users.

Integrating Third-party Libraries. An interesting question is how much of privacy risks are attributed to third-party libraries. We found over 82.2% of apps use at least one third-party libraries: over 55.4%, 22.3%, and 58.5% of apps call libraries related to ads, social networking, and web services. Table 10 shows that over half of the requests are from third-party libraries. This suggests that many data requests are out of developers' control, and they require system-level mechanisms to manage.

LinkedIn (`com.linkedin.android`). Privet rated the Contacts permission of LinkedIn as "MEDIUM (risk)", which indicates it is less suspicious privacy risky of the corresponding permission. In our test, we did seven experiments with different privacy settings (i.e., fully block Contacts, provide only Emails/Phone numbers/names, etc). Our results suggest that LinkedIn can recommend friends by uploading only email addresses. While LinkedIn has a rather ethic privacy policy (e.g., no selling of data), some users may still prefer phone numbers and names not to be exposed to the app. Unfortunately, most modern mobile operating systems offer a rather coarse-grained app-specific ON/OFF switches for such cases.

Eleme (`Ele.me`). Eleme is a popular online-to-offline (O2O) meal ordering app in China. It periodically uploads the current location to retrieve nearby restaurants. When we faked the geo-location coordinates by randomly selecting one within 100m and 500m radius, we observed the restaurant listing changed only slightly. As a result, Privet rates Eleme's location requests as a "MEDIUM (risk)", to indicate that fine-grained localization might not be necessary.

TABLE 10
% of resource requests made by apps and their third-party libraries.

	App	Ads Libs	Other Libs
Network	42.6%	27.8%	29.6%
Account	36.9%	2.2%	60.9%
Geo-location	56.4%	32.8%	10.8%
Net states	50.5%	25.8%	23.7%
Phone states	57.4%	22.6%	20.0%
Others	63.3%	20.0%	16.7%

Notepad (`ru.andrey.notepad`). Notepad is a popular Android app for editing users' notes. Interestingly, Privet rates its Internet permission as "HIGH (risk)". By manually inspecting screenshots and decompiled IR code, we found Notepad uses Internet only for Google Ads advertising, and there is always an advertisement banner on the app page. While advertisement is a gray area to regulate for app distribution providers, as it ties to a long-established business model, Privet currently reports these cases to help end-users make informed decisions.

9 Related Work

UI Control Flow Graphs. Yang et al. [34] propose a callback control-flow graph that shows interactions between user-defined UI event handlers and Android's lifecycle callbacks. And, their follow-up work [35] considers more lifecycle callbacks and window stack states.

However, these efforts cannot handle custom windows/widgets, and they do not consider asynchronous transitions (e.g., due to timers and sensor events). This makes real-world testing difficult to reach these UI elements successfully. Second, they do not consider data flows between these callbacks.

User-facing Tools for Privacy Control. The research community offers more ways to control resource permissions beyond simple warnings. Apex [36] and Dr. Android [37] are tools for enforcing permission-related policies, and they expose Android users to fine-grained permission settings by modifying Android runtime or rewriting app binaries. Rahmati et al. [38] ensure that an app is limited to its minimum set of required permissions with user inputs in the feedback loop. AppFence [7] retrofits the Android runtime to impose privacy controls on apps by shadowing and blocking sensitive data from being exfiltrated off the device. AppIntent [39] presents a sequence of GUI events that lead to sensitive data flows and let analysts decide whether the data flows are intended.

However, without inside knowledge of an app, users can have difficulties in properly configuring tools.

Automated Permission Analysis. Many efforts leverage static code analysis. PScout [40] extracts permission specification from the Android OS source code, and it builds mappings between API calls and permissions. PiOS [9] finds unconsented transmissions of data in iOS apps. Amandroid [41] focuses on computing precise inter-procedural control flow graph. AppProfiler [42] maps patterns of API

calls to predefined (human-readable) behavioral profiles. However, without domain knowledge, profile-based approaches can give generic feedback. In contrast, Privet infers impacts of sensitive data based on user-perceived features.

Taint analysis flags information flows from sensitive data sources to sinks. TaintDroid [10] implements run-time taint analysis for Android, but incurs a 14% performance overhead on a CPU-bound micro-benchmark. AppsPlayground [18] applies static taint analysis to sensitive API and kernel-level monitoring. Unfortunately, taint analysis alone cannot reveal whether an information flow is necessary for user-perceived functionality.

With thousands of daily app submissions, app distribution providers have been trying to shift to automated app vetting process. For example, Google Bouncer [43] automates apps and looks for certain violations of Android Distribution Agreement, rather than unjustified access of resources. However, app distribution providers typically implement random UI exploration, which has poor testing coverage, especially under tight time budget ($<$ 30 min).

Crowdsourced Privacy Analysis. Previous efforts have explored strategies to help users with their disclosure decisions to increase the amount of disclosure without decreased user satisfaction [44], and crowdsourcing has been used for privacy recommendations in tool [45]. Liu et al. [46] demonstrate that a relatively small number of privacy profiles could capture the vast majority of peoples' privacy preferences by a data mining study among 239K real users' app privacy settings. And, they have released a tool called PPA [47]. ProtectMyPrivacy (PMP) [14] crowdsources privacy settings for sensitive data requests per iOS app from over 90K real users. Lin et al. [15] crowdsource a different kind of privacy data – user expectations and reactions on what an app claims to need. These information is then analyzed and made public through PrivacyGrade [33].

As PrivacyGrade analyzes third-party libraries only, it might miss illegal permission usage in app internal code. Moreover, crowdsourcing privacy settings implies that app distribution providers need to first release unvetted apps in the wild, and this is generally against the app store policy. Furthermore, given the vast space of possible privacy settings, effective crowdsourcing requires a sufficient amount of expert user base and time budget. We believe Privet can well compliment these crowdsourcing approaches.

Machine Learning Based Privacy Analysis. Many efforts try to model how benign or malicious apps look like, to find new malicious apps. CHABADA [13] uses the textual app description to estimate what an application should do. AsDroid [48] analyzes the text associated with UI elements. MUDFLOW [12] leverages information flows as the classification feature and learns sensitive data flows from trusted apps to detect malicious abnormal flows. These approaches are trained with different features on a set of apps.

However, the accuracy of these approaches greatly depends on the pre-defined feature set. On the other hand, Privet leverages dynamic analysis to profile run-time app behavior, and uses it as analysis inputs.

10 Discussion and Future Work

Privacy Concerns From Ads. Many app developers are unaware of the kind of information that advertisement providers collect. Interestingly, since advertisement presents a relatively clear end-purpose and is typically delivered by well-established companies, many users can be more lenient in giving up personal data [49]. We acknowledge that this mixed feeling complicates how ads libraries should be considered as privacy risks. Privet currently flags these ads libraries and leave the final decision to human inspectors.

Combinations of Resource Requests. Each test run currently focuses on one resource, and this set up is enough for the purpose of Privet. In the case of multiple data sources being requested, since mobile system APIs typically do not provide overlapping data, manipulating multiple sensitive APIs simultaneously would not reveal additional insights. And, in the case of both a data source and a sink being requested, blocking one can stop the potential privacy risk.

Lack of Sensitive Data Sinks. Some apps do not access sensitive data sinks (or permanent storage media) – for example, while 13 of our apps request for phone state info, they do not issue external I/Os (e.g., storage medium and backend servers) nor inter-process communication (e.g., Android's Intent and Binder). One app developer explained the use of phone state info for the seeding the random number generator. We also observed eight similar apps that request for location info. The "closeness" of these apps implies that they have little chance in leaking user privacy, so Privet skips all apps that do not access sensitive data sinks.

11 Conclusion

This article demonstrates the potential of applying sensitivity analysis to rate app privacy risks. Then, we realize this concept in a system, Privet, for app distribution providers (e.g., Microsoft, Apple, and Google). Our deployment experience suggests that Privet can complement existing tools and human inspectors towards usable privacy that balances privacy protection and app functionality.

References

[1] Forbes, "Your Smartphone Can Photograph You, and Share the Pictures, Without Your Knowledge," 2013.
[2] Geek.com, "How much of your phone is yours?" http://www.geek.com/mobile/how-much-of-your-phone-is-yours-1440611/, 2011.
[3] Privacy Rights Clearninghouse, "Mobile health and fitness applications and information privacy - report to california consumer protection foundation," 07 2013.
[4] Network World, "Obama and Romney Election Apps," 2012.
[5] A. P. Felt et al., "Android Permissions: User Attention, Comprehension, and Behavior," in *SOUPS*. ACM, 2012.
[6] A. R. Beresfod, A. Rice, N. Skehin, and R. Sohan, "MockDroid: Trading Privacy for Application Functionality on Smartphones," in *HotMobile*. ACM, 2011.
[7] P. Hornyack et al., "These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications," in *CCS*. ACM, 2011.
[8] B. Shebaro et al., "IdentiDroid: Android Can Finally Wear Its Anonymous Suit," in *TDP*. ACM, 2014.
[9] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting Privacy Leaks in iOS Applications," in *NDSS*, 2011.

[10] W. Enck et al., "TaintDroid: An Information-Flow Tracking System for Realtime privacy Monitoring on Smartphones," in *OSDI*. USENIX, 2010.
[11] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically Detecting Potential Privacy leaks in Android Applications on a Large Scale," in *TRUST*. SBA Research, 2012.
[12] V. Avdiienko, K. Kuznetsov, A. Gorla, and A. Zeller, "Mining Apps for Abnormal Usage of Sensitive Data," in *ICSE*. IEEE, 2015.
[13] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking App Behavior Against App Descriptions," in *ICSE*. ACM, 2014.
[14] Y. Agarwal and M. Hall, "ProtectMyPrivacy: Detecting and Mitigating Privacy Leaks on iOS Devices Using Crowdsourcing," in *MobiSys*. ACM, 2013.
[15] J. Lin et al., "Expectation and Purpose: Understanding Users' Mental Models of Mobile App Privacy Through Crowdsourcing," in *UbiComp*. ACM, 2012.
[16] A. Saltelli, K. Chan, and E. M. Scott, *Sensitivity Analysis*. Wiley.
[17] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao, "Caiipa: Automated Large-scale Mobile App Testing Through Contextual Fuzzing," in *MobiCom*. ACM, 2014.
[18] V. Rastogi et al., "AppsPlayground: Automatic Security Analysis of Smartphone Applications," in *CODASPY*. ACM, 2013.
[19] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung, "Vision: Automated security validation of mobile apps at app markets," in *MCS*, 2011.
[20] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan, "Automatic and Scalable Fault Detection for Mobile Applications," in *MobiSys*, 2014.
[21] L. L. Zhang, C.-J. M. Liang, W. Zhang, and E. hong Chen, "Towards A Contextual and Scalable Automated-testing Service for Mobile Apps," in *HotMobile*. ACM, 2017.
[22] B. Liu et al., "DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps," in *NSDI*. USENIX, 2014.
[23] SIIS Lab at PSU, "IC3: Inter-Component Communication Analysis for Android," http://siis.cse.psu.edu/ic3.
[24] S. Arzt et al., "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *PLDI*, 2014.
[25] K. Lee et al., "AMC: Verifying User Interface Properties for Vehicular Applications," in *MobiSys*. ACM, 2013.
[26] C. Cortes and V. Vapnik, "Support-vector Networks," in *Machine Learning*. Springer, 1995.
[27] J. Lin, B. Liu, N. Sadeh, and J. Hong, "Modeling Users' Mobile App Privacy Preferences: Restoring Usability in a Sea of Permission Settings," in *SOUPS*. ACM, 2014.
[28] "Genymotion," http://www.genymotion.com/, 2016.
[29] "Androguard," http://github.com/androguard, 2015.
[30] "Apktool," http://ibotpeaches.github.io/Apktool, 2015.
[31] Sauce Labs, "Appium," http://appium.io/, 2016.
[32] rovo89, Tungstwenty, "Xposed," http://repo.xposed.info/module/de.robv.android.xposed.installer, 2016.
[33] "PrivacyGrade," http://www.privacygrade.org/, 2013.
[34] S. Yang et al., "Static Control-Flow Analysis of User-Driven Callbacks in Android Applications," in *ICSE*. IEEE, 2015.
[35] S. Yang et al, "Static Window Transition Graphs for Android," in *ASE*. IEEE, 2015.
[36] M. Nauman, S. Khan, and X. Zhang, "Apex: extending Android permission model and enforcement with user-defined runtime constraints," in *ASIACCS*. ACM, 2010.
[37] J. Jeon et al., "Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android," in *SPSM*. ACM, 2012.
[38] A. Rahmati and H. V. Madhyastha, "Context-Specific Access Control: Conforming Permissions With User Expectations," in *SPSM*. ACM, 2015.
[39] Z. Yang et al., "AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection," in *CCS*. ACM, 2013.
[40] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android Permission Specification," in *CCS*. ACM, 2012.
[41] F. Wei et al., "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps," in *CCS*, 2014.
[42] S. Rosen et al., "AppProfiler: A Flexible Method of Exposing Privacy-Related Behavior in Android Applications to End Users," in *CODASPY*, 2013.
[43] N. Percoco et al., "Adventures in Bouncerland," in *Blackhat*, 2012.
[44] B. P. Knijnenburg et al., "Helping Users with Information Disclosure Decisions: Potential for Adaptation," in *IUI*. ACM, 2013.
[45] M. Bokhorst, "XPrivacy pro," 2013.
[46] B. Liu et al., "Reconciling Mobile App Privacy and Usability on Smartphones: Could User Privacy Profiles Help?" in *WWW*. ACM, 2014.
[47] Bin Liu et al., "Follow My Recommendations: A Personalized Privacy Assistant for Mobile App Permissions," in *SOUPS*. USENIX, 2016.
[48] J. Huang et al., "AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction," in *ICSE*. ACM, 2014.
[49] Ars Technica, "Why Ad Blocking is Devastating to the Sites You Love," http://arstechnica.com/business/2010/03/why-ad-blocking-is-devastating-to-the-sites-you-love/, 2013.

Li Lyna Zhang received the B.Eng degree from University of Science and Technology of China (USTC) in 2013. She is currently a joint Ph.D. candidate of USTC and Microsoft Research. Her research interests include mobile computing, security and privacy, and application testing.

Chieh-Jan Mike Liang received the Ph.D. degree from the Johns Hopkins University in 2011. He is currently a lead researcher at Microsoft Research. His current research interests surround system and networking aspects of sensory and mobile computing.

Zhao Lucis Li received the B.Eng degree from University of Science and Technology of China (USTC) in 2015. He is currently a joint Ph.D. student of USTC and Microsoft Research. His research interests include network and cloud service efficiency.

Yunxin Liu received the Ph.D. degree from Shanghai Jiao Tong University in 2011. His research interests include mobile systems and networking, power management, security and privacy, and human sensing. He is an IEEE senior member.

Feng Zhao received the Ph.D. degree from Massachusetts Institute of Technology. He is currently the CTO and VP for Advanced R&D and Smart Home business at Haier. His research interests include sensor networks, resource management of distributed systems, and mobile and cloud computing. He is an IEEE fellow.

En-Hong Chen received the Ph.D. degree from the University of Science and Technology of China (USTC). He is currently the vice dean of the School of Computer Science. His research interests include data mining and machine learning, social network analysis and recommendation systems. He is an IEEE senior member.