Privacy Challenges in Smart Devices

Katarzyna Olejnik
LCA1, I&C, EPFL

***Abstract*—The number of smart devices around us continues to increase as we enter the era of ubiquitous computing. These devices typically use various sensors, store data about the user, and connect to the Internet. They are also very personal: we bring them around with us, or have them in our homes or workplaces. As a result, these devices pose novel privacy risks. The most prominent example of such a device is the smartphone. Our research goal is to identify these privacy risks and propose solutions, focusing first on smartphones.**

In this proposal, we discuss three existing works and how they relate to our research. We first examine existing issues with smartphone privacy-protection mechanisms. Then we take a look at machine learning techniques which could be used to improve these mechanisms. Finally, we discuss an example of such an application: using machine learning techniques to learn user location privacy policies.

***Index Terms*—privacy, smart devices, Android, mobile, machine learning**

I. Introduction

THE era of smart devices has arrived, and these devices are becoming pervasive on and around us. These devices are called "smart" because they improve on older devices by including new features, and may learn from interacting with a user. Some examples are the Nest thermostat [1], smartwatches such as the Apple watch [2], smart TVs [3], and smartphones. A new trend is the "smart home", where many smart sensing and actuating devices are controlled by a smart hub or smartphone. These devices pose significant privacy concerns because they are extremely personal, and have the ability to collect and send data about the user. Manufacturers include features without considering the privacy implications or properly informing users. Some examples are smart TVs reporting your activity [4] or sending audio samples from voice recognition to the cloud for analysis [5], and Android's and Apple's constant collection of nearby WiFi network information [6].

The devices mentioned above include minimal, if any, privacy controls. At the same time, some smart devices are now enabling installation of third-party apps. This trend will likely continue, with apps being available for installation on most smart devices. Following in the footsteps of Android, the most mature and popular smart device OS, it is likely that a permission system will be introduced on each device.

There are several problems with Android permissions: users don't understand what they mean [7], they are "all-or-nothing", revocation is not supported, they are static, and they are too coarse-grained. Because of coarse-grainedness and poor user understanding, there is a widespread problem of apps requesting more permissions than they need in order to collect user data [8]. In a study conducted by GPEN (Global Privacy Enforcement Network) and published in September 2014, 85% of apps studied did not have a clear privacy policy, and 1/3 appeared to request excessive permissions [9].

Much work has been done to address these limitations and even develop alternatives, but more work is needed [10], [11], [12], [13], [14]. Android has recently introduced changes to the permission system in Android M [15], solving the "all-or-nothing" problem, enabling dynamic permissions, and allowing revocation. *Usability* is still an issue—users must manually configure their permission settings. Previous work has shown that users' decisions about sharing personal information are *context-dependent* [16], [17], [18], [19], [20], [21]. On average, a user interacts with 26.8 apps each month [22]. Thus, configuring 26.8 apps with 5 permissions each [23] and 5 different contexts results in approximately **657 manual decisions**. When considering multiple smart devices, the manual configuration approach becomes even more infeasible. Some recent works address usability issues by crowdsourcing user permission decisions [24], or clustering users into profiles based on their configurations [25]. Such approaches improve usability by lessening user burden, but sacrifice *per-user customization*. Finally, existing solutions only provide binary permission options for users: allow or deny. This provides no tradeoff between privacy and *utility*. For example, if a user is using a weather application, an approximate location will still provide accurate weather, but will not reveal the user's exact location. Thus, the app retains utility, and the user gains some privacy.

A more scalable and user-friendly approach is needed to address the number of access control decisions users will make on smart devices. We propose using machine learning techniques to provide smart privacy protection, with **predictive** capabilities. By modeling, predicting, and responding automatically to app requests for user data, we remove the burden of manual configuration from users and gain *usability*. *Context-awareness* applies naturally to such an approach: contextual information can be supplied to the model as features. The model learned for each user would be *customized* to that user's preferences. Additionally, we aim to go a step further and add **protective** capabilities. Granularity levels for private data provide tradeoffs between privacy and *utility* and serve as a privacy-protection mechanism.

In what follows, we analyze background works in Android, machine learning, and applying machine learning to privacy. In Section II we discuss a work by Enck et al. [26], which concretely shows that the Android permission system is insufficient to protect user privacy and that many apps leak user data. It also provides good background into the technical challenges involved in implementing privacy protections. In Section III we discuss a work by Lewis and Catlett [27], which proposes a way to improve a standard machine learning

technique, uncertainty sampling. Uncertainty sampling is used to train an accurate classifier with minimal manual labeling of instances. Using such an approach would minimize user burden, which is key for usability. In Section IV we discuss a work by Cranshaw et al. [28], which demonstrates how machine learning can be used to learn user location privacy policies. This work is important because the authors apply machine learning to privacy, albeit at a smaller scale than we intend to do. Finally, in Section V we discuss our current research, particularly efforts to develop a prototype Android app providing smart permission management—SmarPer.

SmarPer is an implementation of the ideas we have been discussing so far. SmarPer intercepts app requests for user data at runtime and allows the user to decide whether to deny, allow, or *obfuscate* access to their private data. Obfuscation reduces the level of detail available to the requesting app; it is an implementation of both a privacy/utility tradeoff and a privacy-protection mechanism. SmarPer learns how the user responds over time based on some manual decisions and contextual information; it introduces *context-aware* and *automatic* decisions via machine learning, and creates a *customized* model for each user.

II. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones

In this section we discuss the work by Enck et al. [26]. This work is seminal in the field of Android privacy and provides a concrete motivation for our thesis, demonstrating the issues with Android permission management, and the technical challenges involved in implementing privacy protections.

A. The problem

The Android permission system has several shortcomings, resulting in users having *low visibility* into how their data is used. After a user agrees to install an app, the app can use the permissions it requested with any frequency, with no further notification to the user. The app can also exfiltrate user data without user knowledge. These issues have been partly addressed since then, in the new version of Android announced in May 2015, which introduces dynamic, runtime permissions [15]. Enck et al. aim to address these issues and create a tool to provide users with greater visibility into how Android apps access their data. A key challenge is identifying what information is leaving the device, and where is it being sent.

B. Their solution

Taint tracking, or *dynamic taint analysis*, refers to marking—or *tainting*—sensitive information in the system, and tracking whether it is exposed (reaches a network interface) or affects other data in the system. Existing approaches to taint tracking use instruction-level instrumentation or whole system emulation, both of which incur serious slowdowns [29], [30], [31]. Implementing taint tracking for the x86 instruction set has proven challenging, particularly in avoiding false positives and false negatives [32], [33]. Enck et al. create a realtime system-wide taint tracking modification to the Android OS (TaintDroid). TaintDroid tracks private data as it moves through the Android OS, from taint source (e.g., sensor), through trusted applications (pre-installed on the phone), to untrusted applications (installed by the user), and finally to taint sink (e.g., network interface). Outgoing connections are flagged if they are suspected to contain private data. In this section, we describe the TaintDroid implementation.

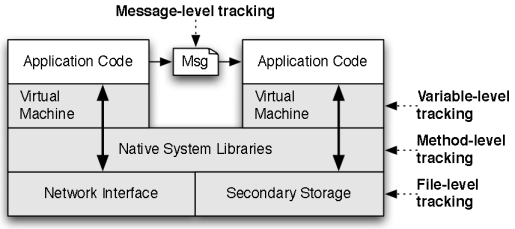

Fig. 1. TaintDroid architecture [26]. This figure illustrates the four areas of the Android OS where the authors implemented taint tracking: IPC messages, interpreter variables, native methods, and files.

Taint markings and taint tags: The authors introduce the following terminology. A *taint tag* is a set of *taint markings* from the universe L of all possible taint markings on the system. Two examples of taint markings are location and phone number. The authors add taint tags for four areas of the Android OS, shown in Figure 1. Each of those objects has an associated *taint tag*, which can include multiple *taint markings*. Taint tags can be empty (i.e., no tainted information is inside). As an example, a variable in the Dalvik interpreter which contains a GPS coordinate obtained from Android's LocationManager will have the "location" *taint marking* in its *taint tag*.

Taint sources and sinks: The authors instrument the following taint sources, which deliver the appropriate *taint markings*, shown in parentheses, upon access: *low-bandwidth sensors* via the associated sensor manager (location, sensors); *high-bandwidth sensors* via data buffers and files (microphone, camera); *information databases* via the associated files (contacts); and *device identifiers* via the associated APIs (phone number, ICC-ID, IMEI). The network interface is configured as the only taint sink.

Taint tracking and taint tag storage: The authors introduce taint tracking and taint tag storage in Android OS, in the four areas shown in Figure 1. The majority of taint-tracking occurs during execution of *interpreted code*. For this purpose, the authors instrument the Dalvik interpreter. Dalvik provides variable semantics—which distinguish pointers from data values—and thus makes it easy to identify what needs to be tracked. This avoids the issues present in other taint tracking systems, such as accidental tainting of the stack pointer [32]. The authors introduce taint-propagation rules for relevant Dalvik opcodes. Taint tags are stored adjacent to the relevant values in registers, providing spatial locality.

Taint propagation and storage for *native code* is implemented manually: the authors provide a taint propagation mapping (i.e., whether taint tags should propagate from method arguments to method return value) for certain methods. For

the rest, the authors implement a heuristic.

For *IPC messages*, the authors propagate taint tags at the message level. Tainting at the variable level here could be subverted by a malicious app by unpacking the parcel in a non-standard way. They note that tainting at the message level can introduce false positives.

Finally, for *secondary storage*, the authors propagate taint tags at the file level. Once again, the authors note that this could introduce false positives, but accepted this tradeoff for an increase in performance.

C. Results

App study: To evaluate TaintDroid, the authors tested 30 popular apps and reported on the information leaks discovered. The 30 apps were selected from a 2010 survey of the 50 most popular Android apps. All of the apps required the Internet permission, plus at least one other permission to access private data: phone state, location, camera, or audio. The authors installed these apps on a phone with TaintDroid, and then manually used the apps for approximately 100 minutes.

They discovered abuse of personal data by apps, with only a few apps displaying good practices. Two thirds of the apps handled user data suspiciously. Two apps revealed phone number, IMSI, and ICC-ID; nine apps revealed IMEI, and 15 apps revealed user location to advertisement servers. Bad practices discovered include apps sending data to remote servers prior to the first run of the app, apps using the IMEI as a client ID, apps not displaying an end-user license agreement (EULA) stating what information is collected, or apps displaying an inaccurate EULA. Two apps used good practices: one displayed a privacy notice stating that the app collects the IMEI, and the other collects only the hash of the IMEI. TaintDroid did not produce any false positives.

Benchmarks: To evaluate the overhead of TaintDroid and potential impact on the user experience, the authors conduct several benchmark tests. The first test is a macrobenchmark. Here the authors compare the amount of time it takes to conduct certain operations on vanilla Android vs TaintDroid. The operations tested are app load time, creating a contact in the address book, reading from the address book, placing a phone call, and taking a picture. Increase in overhead for TaintDroid ranged from 3% to 29%.

The next test was a Java microbenchmark: an Android port of CaffeineMark 3.0. The results exhibited some variation across the tests: the greatest difference in performance was with the "string" benchmark (approx. 20% overhead). The authors attribute this to their heuristic for native methods. Overall, TaintDroid incurs a 14% overhead compared to vanilla Android, and a 4.4% memory overhead.

Finally, the authors conduct an IPC benchmark, to quantify overhead on IPC (i.e., Binder messages). They implemented simple client and service applications which perform Binder transactions as quickly as possible. The results show that TaintDroid incurs a 27% overhead compared to vanilla Android, and a 3.5% memory overhead.

D. Discussion

To our knowledge, this is the first work to quantify app information leakage on Android. The authors contribute the first taint tracking system for a mobile phone.

With regards to our thesis, this work demonstrates the issues with Android permissions. It also provides background into the technical aspects of implementing privacy protections on Android, as well as the Android OS itself. Implementing taint-tracking, on Android or other smart devices, is *insufficient to improve privacy protection*: it provides only *monitoring*, not *mitigation* capabilities. Although TaintDroid flags outgoing connections, these must be *confirmed manually* by an expert. Information could be sent out of the device as a result of an informed user action; for example, getting weather based on the current location. Such instances do not constitute information leakage. We could envision combining SmarPer with TaintDroid, to determine if an app will send data off the device after access. However, this type of approach may introduce performance issues.

The authors mention themselves that there are several limitations to TaintDroid, specifically in the approach taken, the implementation, and in the taint sources used. They make no guarantees or proofs about completeness of private data tracked. A design limitation is that TaintDroid only tracks data flows, as opposed to control flows, for performance reasons. Implementation-wise, certain objects which are created in the native address spaces are untracked.

III. Heterogeneous Uncertainty Sampling for Supervised Learning

In this section we discuss the work by Lewis and Catlett [27]. This work proposes an improvement to the usability and speed of the uncertainty sampling technique proposed in a previous paper by Lewis and Gale [34]. Uncertainty sampling is a machine learning technique used to minimize the number of instances a user must label as positive and negative for training a binary classifier. It is employed when large amounts of unlabeled data are available in order to minimize the amount of labeling the user must do. In fact, it is a technique used by Fang et al. [35] to automatically configure user privacy settings on online social networks with minimal user input, which drew us to this work.

In the context of our thesis, minimizing the amount of required user input to train a classifier decreases user burden and thus improves usability. We envision potentially incorporating such a technique into our novel privacy protections for smart devices. Additionally, smart devices are resource-constrained platforms, so the techniques we use need to be fast and have minimal impact on the user experience.

A. The problem

The basic uncertainty sampling procedure is shown in Figure 2. Of particular interest is step 2.d: re-training a classifier on all newly labeled instances. If the classifier training procedure is resource-intensive or time-consuming, this may become a barrier to using uncertainty sampling in practice. In this work, the authors aim to make uncertainty sampling more usable in practice by addressing the repeated training phase.

1. Obtain an initial classifier
2. While expert is willing to label instances
   (a) Apply the current classifier to each unlabeled instance
   (b) Find the b instances for which the classifier is least certain of class membership
   (c) Have the expert label the subsample of b instances
   (d) Train a new classifier on all labeled instances

Fig. 2. The procedure for uncertainty sampling [27]. A binary classifier (in this case) is trained on all labeled instances. The instances of which the classifier is least sure about—i.e., $P(C_1)$ and $P(C_2)$ are close to 0.5—are presented to a human to label and added to the training set, then the process repeats.

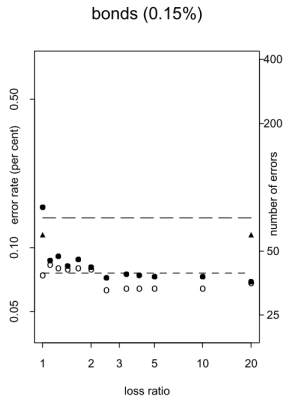

Fig. 3. The results for the "bonds" category [27]. The 0.15% value indicates the percent of positive instances in the training set—this value is small for document classification tasks. Black dots represent the set of 299 uncertainty samples, white dots represent the set of 999 uncertainty samples, long dashes represent the set of 1,000 random samples, and short dashes represent the set of 10,000 random samples. The error rates for the uncertainty samples and the set of 10,000 random samples are comparable.

B. Their solution

The authors propose a "heterogeneous" approach to uncertainty sampling: they modify uncertainty sampling to use two classifiers. In the context of Figure 2, steps 1 and 2 are implemented with an efficient classifier (i.e., training phase is cheap). *Step 3 is added*: use the generated uncertainty samples to train your final classifier, the one you want to use for the classification task. Thus, if the training phase for this classifier is time-consuming or resource-intensive, it only needs to be done once instead of several times, resulting in performance improvements.

Document classification task, constraints: The authors target binary document classification as their application of interest. This informs several choices and restrictions. First, texts for document classification typically reside in large databases which support boolean queries. As a result, they choose C4.5 as their final classifier, to be used in the new third step. C4.5 produces a decision tree, and can also produce decision rules, which are easy to translate into boolean queries. Next, the authors chose a dataset which mimics many of the properties seen in document classification tasks: few instances are positive, the classes are noisy, and the classification cannot be perfectly determined from the text in the document. The authors use a dataset of 371,454 *newspaper article titles* and associated *subject categories*. The authors choose a subset of 10 categories from this dataset ("bonds", "dukakis", etc). Each word in the title becomes a binary attribute, resulting in 67,331 attributes. The categories mirror human labeling of instances. Finally, the data is extremely sparse, so the classifier used for uncertainty sampling should be able to handle sparse data easily.

Modifications to C4.5: The authors make custom modifications to C4.5 to implement heterogeneous uncertainty sampling. In binary document classification, the prior probability of a class $P(C_1)$ is typically extremely small. However, the authors had decided on C4.5 which does not support priors. Instead, the authors introduce a *loss ratio* to specify the cost of false positives and false negatives. A loss ratio greater than 1 indicates that false positives are more costly than false negatives.

Probabilistic classifier: For steps 1 and 2 of uncertainty sampling, the authors use a highly efficient probabilistic classifier of their own design, first described in the work by Lewis and Gale [34]. Importantly, this classifier can easily handle sparse data. The classification task is binary, with two classes, C_1 and C_2. The classifier uses the following estimate for the probability that an instance $\mathbf{w}$, which consists of a series of words w_i, belongs to class (i.e., category) C_1:

$$P(C_1|\mathbf{w}) = \frac{\exp(a + b\sum_{i=1}^{d} \log \frac{P(w_i|C_1)}{P(w_i|C_2)})}{1 + \exp(a + b\sum_{i=1}^{d} \log \frac{P(w_i|C_1)}{P(w_i|C_2)})}$$

Importantly, the authors make the assumption that the features (words) are independent. This is not a correct assumption for natural language. To correct for this, they weight the sum of the log probabilities by the parameter b. The parameter a represents the quantity $\frac{P(C_1)}{1-P(C_1)}$. In the context of document classification, $P(C_1)$ is hard to estimate from the data since it is very small (i.e., few instances are positive). This problem is compounded since the training data is not a random sample.

This classifier is trained via the following procedure: The values $P(w_i|C_1)$, $P(w_i|C_2)$, and $P(w_i)$ are estimated for every w_i. w_i's with large values of $P(w_i) \times \log P(w_i|C_1)/P(w_i|C_2)$ were chosen as features. $\prod_{i=1}^{d} \frac{P(w_i|C_1)}{P(w_i|C_2)}$ is computed for each instance, and then a and b are determined by logistic regression.

C. Results

The authors conducted a series of tests to evaluate heterogeneous uncertainty sampling. Specifically, the authors test if 1.) heterogeneous uncertainty sampling produces accurate classifiers with less training instances than random sampling, 2.) the effect of varying the loss ratio, and 3.) the effect of using different classifiers for uncertainty sample selection and the final classification. It was infeasible to use C4.5 for both, so the authors instead test using the probabilistic classifier for both.

The authors conducted 10 uncertainty sampling trials on their 10 chosen subject categories. Each trial started with providing the probabilistic classifier with 3 initial positive

instances. Uncertainty sampling with a subsample size of 4 proceeded for 249 rounds. From each trial, the authors selected groups of uncertainty samples of size 299 and 999 (after sampling for 74 rounds and 249 rounds, respectively). These groups of uncertainty samples were then used to train C4.5 rules, with the loss ratio varying from 1 to 20. To compare to random sampling, the authors trained C4.5 rules on sets of 1,000 and 10,000 random samples (technically 997 and 9,997; the initial 3 positive instances were not random). The results for the "bonds" category are shown in Figure 3.

The authors also compare the average percent error and standard deviation for uncertainty samples of size 999 and random samples of size 10,000, both when using C4.5 and the probabilistic classifier for the final classification, in Table 2 of the paper. As a baseline, the authors compare the performance of both to a dummy classifier called "Reject All", which classifies all instances as negative—the *accuracy* of which may be satisfactory, given that in this scenario most instances are negative. Table 3 of the paper shows number of false positives and false negatives for the same data.

D. Discussion

In general, heterogeneous uncertainty sampling performed well. Uncertainty sampling with 999 instances produced classifiers with accuracy comparable to classifiers trained on 10,000 random instances. With a loss ratio of 5, uncertainty samples performed significantly better (p=.03) than the random samples. Loss ratios between 3 and 20 appeared to produce accurate classifiers. The authors state that these results indicate heterogeneous uncertainty sampling can indeed be effective.

In the context of our thesis, these results are promising. They indicate that a heterogeneous approach to uncertainty sampling still yields accurate classifiers, and requires less training data (and thus, user labeling) than random sampling. Thus, using such an approach when devising novel privacy protection mechanisms for smart devices looks feasible. Minimizing user input is key for reducing user burden and maximizing usability, and efficiency is crucial for resource-constrained platforms. A visible slowdown in the device could also frustrate the user, so we would like to avoid this as much as possible.

Our main critique of this paper is that the authors' approach could be more robust. It is understandable that the authors were constrained by their tools—for instance, C4.5 does not support priors, so the authors introduced a loss ratio instead. The authors' probabilistic classifier makes some invalid independence assumptions in the context of natural language, which the authors acknowledge and counterbalance by setting the parameter b by logistic regression. We are skeptical of whether the authors' classifier will always provide accurate estimates. Indeed, the authors themselves state in their previous paper [34] that the classifier appears to work well for text categorization, but that they have not compared the performance to other, more complex classifiers.

IV. User-Controllable Learning of Location Privacy Policies with Gaussian Mixture Models

In this section, we discuss the work by Cranshaw et al. [28]. The authors model evolving user location privacy policies with Gaussian mixture models, while still letting the user audit the model's decisions. In the context of our thesis, this work shows a concrete example of how machine learning techniques can be applied to model and predict user privacy preferences over time. Additionally, the authors make a great effort to maximize usability through gradual model evolution, and giving the user the power to view and change the built model. These are also important considerations for SmarPer.

A. The problem

Location-based services are moving from a check-in model to a continuous tracking model, in order to better facilitate spontaneous encounters, or offer recommendations about new places to visit. Additionally, there are now health-tracking applications which also continuously monitor the user. These types of continuous tracking services pose privacy risks. At the same time, privacy controls are limited and coarse-grained. The user is left with the task of specifying a complex sharing policy with coarse-grained controls, which is not usable and frustrating.

B. Their solution

The authors propose using user-controllable Gaussian mixture models to learn user sharing preferences by creating a mapping between the user's policy and the model. A Gaussian mixture model combines multiple Gaussian distributions to model data. The goals of the authors are to maximize usability: the user should be in control of the generated model at all times, and the model should evolve gradually as new data arrives, to maximize understandability. The authors decide to use a generative model, because these models are easier to extend to the semi-supervised learning approach, which would improve accuracy by leveraging unlabeled data and thus further decrease user burden.

Formalizing location-sharing: Gaussian mixture models appear to be a natural model for location-sharing in the Locaccino [1] continuous-tracking application. An *observation* x of a user is defined as a point (lat, lon, t) in a three-dimensional space. Users create *location rules* per friend which specify under which conditions their friend can see their location. *Location rules* can be based on latitude and longitude, time, or both.

A location rule, meanwhile, can be represented as a cuboid in the same three-dimensional space. If an observation falls into one of the "allow" rule cuboids, the user's location will be revealed. Thus, to create a machine learning model, the authors slightly relax the previous concepts and use Gaussian mixture models with axis-aligned Gaussians. Each Gaussian in the mixture model corresponds to one location rule. The authors create a Gaussian mixture classifier which consists of two Gaussian mixture models: one for deny rules, and one for

[1] http://www.locaccino.org

For j rounds:

1) Add some data to training set.
2) Train the NewEachRound model, M_j^*, on the training data.
3) Compute symmetrized KL-divergence between the Gaussians in $\theta_{i,j-1}$ and $\theta_{i,j}^*$, represented as $D[k_i, k_i^*]$ for arbitrary Gaussians k_i and k_i^*.
4) If the number of Gaussians is equal for either the deny models, the allow models, or both: create a mapping of Gaussians between $\theta_{i,j-1}$ and $\theta_{i,j}^*$.
   - Entries in D are weights in a bipartite graph, on which the minimum cost matching is computed.
   - Result is a mapping m, where $m[k_i]$ gives the Gaussian k_i^* to which k_i is mapped.
5) Conduct a local operation: add, $swap$, or $delete$ a Gaussian, to create M_j by moving M_{j-1} closer to M_j^*.
6) Evaluate both M_j and M_j^* on the test data.

Fig. 4. The basic training and evaluation procedure. The model M_j^* informs the gradual evolution of the model M_{j-1} at each round into M_j. KL-divergence is used as a metric for determining the difference between the two models.

allow rules. Classification is based on which model provides the higher maximum likelihood estimate.

Training, model evolution, and testing procedure: The authors train and evaluate models in rounds, since they are interested in gradual model evolution. Each round begins with adding some new data to the training set. The authors maintain two Gaussian mixture classifiers, one which evolves gradually, and one which is trained anew at each round. They conduct one local operation at each round to move the gradual model towards the freshly-trained model, and then evaluate the performance of both on the test data. The specific steps of the procedure are shown in Figure 4. The authors introduce the following notation: a Gaussian mixture classifier M at the round j is denoted as $M_j = (\theta_{0,j}, \theta_{1,j})$, indicating the mixture models for "deny" and "allow", respectively.

The authors implement four algorithms to test gradual model evolution. We describe the first algorithm, *MaxMatchedKL*, in detail and then state how the other algorithms differ. The three local operations available to these algorithms are to add, $delete$, or $swap$ Gaussians from M_j^* into M_{j-1} to create M_j. First, some notation: K_i^*, K_i represent the number of Gaussian components in M_j^* and M_{j-1}, respectively. *MaxMatchedKL* will add a Gaussian if $K_i^* > K_i$, delete if $K_i^* < K_i$, and swap if $K_i^* = K_i$. The Gaussian chosen to add is given by computing $\max_{k_i} D[k_i, k_i^*]$ for each Gaussian in $\theta_{i,j}^*$. For deletion, the authors compute $\max_{k_i^*} D[k_i, k_i^*]$ for Gaussians in $\theta_{i,j-1}$. To swap, the authors maximize instead over matched Gaussians in the mapping m, and choose $\arg\max_k(D[k, m[k]])$. The change to the model at this round is only accepted if it will increase the likelihood given the data. *MaxMatchedWeightedKL* introduces probabilistic adding, deleting, swapping. When choosing Gaussians to add, delete, or swap, they are weighted by their relative importance to the mixture model. *RandomMatchedKL* is a randomized version of *MaxMatchedKL*, and likewise *RandomMatchedWeightedKL* is a randomized version of *MaxMatchedWeightedKL*.

C. Results

The authors collected a dataset of user-specified location-sharing policies and observations from Locaccino. They sampled 2,000 location observations from each user, uniformly, without replacement. These were split into a training and testing set: the first (temporal) 60% of the observations were used for training, and the last 40% as testing. Evaluating an unlabeled observation on the policy of each user gives the true classification.

Gaussian mixture models are trained and evaluated for 30 rounds, using the procedure described in the previous section. At each round, the next 75 labeled observations are added to the training data, and the model is evaluated on the test data. The test data remains the same for each round. Importantly, Locaccino policies (and subsequently, the classifiers trained) are *per-friend*, so even though the authors only had data from 11 users, they had 88 location-sharing policies to test for each of their four algorithms.

The authors then plot the accuracy of their four algorithms and compare the accuracy to the *NewEachRound* classifier. After 30 rounds, the average accuracy of *NewEachRound* was 0.87, whereas the average accuracies for the user-controllable models ranged from 0.85 to 0.76. They also chart the mean KL-divergence between rounds for each model, i.e. how much the model changed each round, along with the standard deviation. The mean KL-divergence for *NewEachRound* was 2.15 bits per round, and for the other models this ranged from 0.97 to 1.49.

D. Discussion

This work demonstrates that user location-sharing policies (from Locaccino, specifically) can be modeled and predicted accurately by machine learning algorithms. It also shows that models which are restricted evolve incrementally can be similarly accurate to models which are not.

In the context of our thesis, this work shows an example of how machine learning techniques can be applied to learn user privacy policies, and contributes some important usability aspects. Models which evolve incrementally are more intelligible to the user, and the authors have shown that such models do not suffer a drop in accuracy. Importantly, the user should also have input into the model. If the user has no visibility or control over the generated model, this also becomes a usability issue. However, the authors do not test the user-controllable aspect of their work. In addition, with our thesis we plan to go a step further: in addition to smart prediction and modeling of user preferences, we will include additional granularity levels for private data (a privacy/utility tradeoff).

The selection of testing and training data by the authors could be improved. First, the authors sample from user observations with no temporal restrictions. A better approach would be to use the first 2,000 observations of the user. Second, the test data remains the same throughout the experiment (last 60% of observations). The test data could be a sliding window of the next temporal set of user observations. Additionally, more in-depth evaluation of results would be helpful. The authors present only accuracy for each of their four algorithms.

False positives and false negatives could be included as well, and an analysis of relative cost of each error to the user—false positives (sharing location erroneously) should probably be a more costly error.

V. Research Proposal

Smart devices pose novel privacy risks, and include little to no privacy controls for the user. We have begun to see the emergence of apps for smart devices other than smartphones. Eventually, as devices become more sophisticated and mature, they may go the way of Android and provide privacy protections via permissions. Users will be left with the problem of manually configuring their preferences across all devices with coarse-grained controls. We believe that machine learning could be used to provide smart privacy features for such devices. We begin our investigation with Android, the most mature smart device OS: much work has been done on privacy protections for Android, but usability and utility issues remain. In the previous sections, we discussed three papers which provide the foundation for our first research direction: creating *usable privacy protection mechanisms for Android via machine learning*.

With regard to this first direction, we have built a smart permissions prototype for Android, SmarPer. SmarPer introduces usable, context-aware, fine-grained, and automatic privacy protections for permission decisions. SmarPer learns and predicts user permission decisions from context and a handful of manual user permission decisions at runtime. Learning the user's preferences and responding automatically decreases user burden and increases *usability*. SmarPer also provides privacy protection capabilities in the form of a *privacy/utility tradeoff*. Implementing this tradeoff is challenging, and requires deep knowledge of Android OS. In addition, the user can view and audit decisions made automatically by SmarPer. SmarPer builds on previous work from LCA1, SPISM [36], which introduced semi-automatic, context-aware information sharing decisions for instant messaging applications. We have developed a working SmarPer prototype, based on the open source project XPrivacy [37]. SmarPer currently includes dynamic permission decisions at runtime, finer-grained permission decisions, additional granularity levels for private data access (privacy/utility tradeoff), and logging of decisions and contextual information in a database. Our next steps for this direction are quality assurance and running a data collection campaign. We will use the collected data to inform our choice of machine learning technique for modeling and predicting future user decisions. After development of an initial model we will run another campaign, this time to evaluate the effectiveness of our predictive model.

Our next direction is to *evaluate how Android applications access user data*. There has been a great deal of work on how to identify malicious Android applications, or applications which behave suspiciously [38], [39], [40], [41]. However, there is little work on analyzing how benign applications access user data over time [42], [43], [21]. For example, how many requests for user data do apps make in the background, when the user is not using the app? Are any requests made when the user is sleeping and not using their phone? Users have little visibility into how apps access their data. This type of analysis will demonstrate the privacy risk from benign Android applications.

A complementary analysis will follow, to *analyze how users are making use of existing permission mechanisms* (for example with AppOps or LBE Privacy Guard, which allow permission revocation after installation). Such an analysis has been hinted at in [25], but their work focused on predicting how users will configure their settings. Some questions that could be answered by such an analysis are: how many users revoke a permission? Do users change their settings often, perhaps based on context? Do users like this type of permission manager better than one which offers less control, or do they find it confusing? A study like this could provide additional insight into developing novel privacy protection mechanisms.

Finally, we will move to analyze privacy risks of other smart devices, starting with the *smart hub in the smart home*. A smart hub, such as Apple's HomeKit [44] or the SmartThings Hub [45], provides centralized control over all of a user's smart devices, either via a physical device or on a smartphone. Eventually, all of the connected devices and the hub may use permissions, or some other access control mechanism. We believe our Android smart permission techniques could also be applied to this space.

References

[1] "Home — Nest," https://www.nest.com, accessed June 2015.

[2] "Apple - Apple Watch," http://www.apple.com/watch/, accessed June 2015.

[3] "Samsung Smart TV - TV Has Never Been This Smart," http://www.samsung.com/us/experience/smart-tv/, accessed June 2015.

[4] C. Arthur, "Information commissioner investigates LG snooping smart TV data collection," http://www.theguardian.com/technology/2013/nov/21/information-commissioner-investigates-lg-snooping-smart-tv-data-collection, accessed June 2015.

[5] "Not in front of the telly: Warning over 'listening' TV," http://www.bbc.com/news/technology-31296188, accessed June 2015.

[6] Julia Angwin, Jennifer Valentino-DeVries, "Apple's iPhones and Google's Androids Send Cellphone Location - WSJ," http://www.wsj.com/articles/SB10001424052748703983704576277101723453610, 2011, accessed April 2015.

[7] A. Felt, E. Ha, S. Egelman, and a. Haney, "Android permissions: User attention, comprehension, and behavior," in *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, 2012.

[8] N. Bilton, http://bits.blogs.nytimes.com/2012/02/12/disruptions-so-many-apologies-so-much-data-mining/, accessed January 2015.

[9] Information Commissioner's Office of the UK, "Global survey finds 85% of mobile apps fail to provide basic privacy information — ICO," https://ico.org.uk/about-the-ico/news-and-events/news-and-blogs/2014/09/global-survey-finds-85-of-mobile-apps-fail-to-provide-basic-privacy-information/, accessed January 2015.

[10] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi, "ASM: A Programmable Interface for Extending Android Security," in *Proceedings of USENIX Security Symposium*, 2014.

[11] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical Policy Enforcement for Android Applications," in *Proceedings of USENIX Security Symposium*, 2012.

[12] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "Appguard enforcing user requirements on android apps," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, 2013, vol. 7795, pp. 543–548.

[13] "Welcome to Blackphone," https://www.blackphone.ch/, accessed January 2015.

[14] "PrivacyGrade," http://www.privacygrade.org, accessed June 2015.
[15] "Permissions — Android Developers," http://developer.android.com/preview/features/runtime-permissions.html, accessed June 2015.
[16] M. Benisch, P. G. Kelley, N. Sadeh, and L. F. Cranor, "Capturing location-privacy preferences: Quantifying accuracy and user-burden tradeoffs," *Personal and Ubiquitous Computing*, vol. 15, pp. 679–694, 2011.
[17] N. Sadeh, J. Hong, L. Cranor, I. Fette, and P. Kelley, "Understanding and Capturing People s Privacy Policies in a Mobile Social Networking Application," *Personal and Ubiquitous Computing,*, vol. 13, pp. 401–412, 2009.
[18] I. Smith, S. Consolvo, and A. Lamarca, "Social disclosure of place: From location technology to communication practices," in *Proceedings of International Conference on Pervasive Computing (Pervasive)*, 2005.
[19] E. Toch, J. Cranshaw, P. H. Drielsma, J. Y. Tsai, P. G. Kelley, J. Springfield, L. Cranor, J. Hong, and N. Sadeh, "Empirical models of privacy in location sharing," in *Proceedings of ACM International Conference on Ubiquitous Computing (Ubicomp)*, 2010.
[20] H. Wu, B. P. Knijnenburg, and A. Kobsa, "Improving the prediction of users' disclosure behavior. . . by making them disclose more predictably?" in *Proceedings of International Symposium on Usable Privacy and Security (SOUPS)*, 2014.
[21] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov, "Android Permissions Remystified: A Field Study on Contextual Integrity," 2015.
[22] "Smartphones: So Many Apps, So Much Time," http://www.nielsen.com/us/en/insights/news/2014/smartphones-so-many-apps--so-much-time.html, accessed January 2015.
[23] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, "1,000,000 Apps Later: A View on Current Android Malware Behaviors," in *International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.
[24] Y. Agarwal and M. Hall, "ProtectMyPrivacy: detecing and mitigating privacy leaks on iOS devices using crowdsourcing," in *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2013.
[25] B. Liu, J. Lin, and N. Sadeh, "Reconciling mobile app privacy and usability on smartphones: could user privacy profiles help?" in *Proceedings of International Conference on World Wide Web (WWW)*, 2014.
[26] W. Enck, P. Gilbert, 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 *Proceedings of the Usenix Symposium on OS Design and Implementation (OSDI)*, 2010.
[27] D. Lewis and J. Catlett, "Heterogeneous uncertainty sampling for supervised learning," in *Proceedings of International Conference on Machine Learning (ICML)*, 1994.
[28] J. Cranshaw, J. Mugan, and N. Sadeh, "User-Controllable Learning of Location Privacy Policies with Gaussian Mixture Models," in *Proceedings of American Association for the Advancement of Artificial Intelligence (AAAI)*, 2011.
[29] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis," in *Proceedings of ACM conference on Computer and Communications Security (CCS)*, 2007.
[30] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding Data Lifetime via Whole System Simulation," in *Proceedings of USENIX Security Symposium*, 2004.
[31] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand, "Practical taint-based protection using demand emulation," p. 29, 2006.
[32] A. Slowinska and H. Bos, "Pointless Tainting? Evaluating the Practicality of Pointer Tainting," in *Proceedings of European Conference on Computer Systems (EuroSys)*, 2009.
[33] Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "Privacy scope: A precise information flow tracking system for finding application leaks," *Tech. Rep. EECS-2009-145, Department of Computer Science, UC Berkeley.*, 2009.
[34] D. D. Lewis and W. A. Gale, "A Sequential Algorithm for Training Text Classiers," in *Proceedings of International Conference on Research and Development in Information Retrieval (ACM-SIGIR)*, 1994.
[35] L. Fang and K. LeFevre, "Privacy wizards for social networking sites," in *Proceedings of International Conference on World Wide Web (WWW)*, 2010.
[36] I. Bilogrevic, K. Huguenin, B. Agir, M. Jadliwala, M. Gazaki, and J.-P. Hubaux, "A machine-learning based approach to privacy-aware information-sharing in mobile social networks," *Pervasive and Mobile Computing*, no. 17, 2015.
[37] M. Bokhorst, https://www.github.com/M66B/XPrivacy, accessed January 2015.
[38] S. Chakradeo, B. Reaves, and W. Enck, "MAST: Triage for Market-scale Mobile Malware Analysis," in *Proceedings of ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2013.
[39] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective Real-time Android Application Auditing," in *Proceedings of IEEE Security and Privacy (S&P)*, 2015.
[40] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," 2013.
[41] W. Xu, F. Zhang, and S. Zhu, "Permlyzer: Analyzing Permission Usage in Android Applications," in *Proceedings of International Symposium on Software Reliability Engineering (ISSRE)*, 2013.
[42] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Profiledroid: Multi-layer profiling of android applications," in *Proceedings of International Conference on Mobile Computing and Networking (Mobicom)*, 2012.
[43] M. Frank, B. Dong, A. P. Felt, and D. Song, "Mining permission request patterns from Android and Facebook applications," in *Proceedings of IEEE International Conference on Data Mining (ICDM)*, 2012.
[44] "HomeKit - Apple Developer," https://developer.apple.com/homekit/, accessed June 2015.
[45] "Smart Things — Smart Home. Intelligent Living." http://www.smartthings.com/, accessed June 2015.