Privacy Policies Change Management for Smartphones

Debmalya Biswas
Nokia Research Center, Lausanne, Switzerland
debmalya.biswas@nokia.com

Abstract—The ever increasing popularity of apps stems from their ability to provide highly customized services for the user. The flip side is that to provide such customized services, apps need access to very sensitive personal user information. This has led to a lot of rogue apps that e.g. pass personal information to 3rd party Ad servers in the background. Studies have shown that current app vetting processes which are mainly restricted to install time verification mechanisms are incapable of detecting and preventing such attacks. We argue that the missing fundamental aspect here is the inability to capture and control run-time characteristics of apps, e.g. we need to know not only the list of sensors that need to be accessed by an app but also their frequency of access. This leads to the need for an expressive policy language that in addition to the list of sensors, also allows specifying when, where and how frequently can they be accessed.

An expressive policy language has the disadvantage of making the task of an average user more difficult in setting and analyzing the consequences of his privacy settings. Further, privacy policies evolve over time. Over time, users are likely to change their privacy settings, as a response to a recently discovered vulnerability, or to be able to install that “much desired” app, etc. Such a policy change affects both already installed (may no longer be compliant) and previously rejected apps (may be compliant now). In this paper, we propose an integrated privacy add-on that (i) compares the apps profiles vs. user’s privacy settings, outlining the points of conflict as well as the different ways in which they can be resolved. And (ii) provides efficient change management with respect to any changes in user privacy settings.

Index Terms—Privacy policies; Conflict detection; Policy evolution; Smartphone apps/services;

I. INTRODUCTION

Apps (application for smartphones) are the lifeline of today’s smartphones. Their immense popularity is evident from the thousands of apps available in Nokia’s Ovi Store, Apple’s AppStore, Google’s Android Marketplace, Windows Marketplace Hub, etc. The reason apps are so popular and useful is because they provide services highly customized to different aspects of our life, from recommending location based places of interest to monitoring our health. The downside is of course that to do so apps collect a lot of personal information about the user. This personal information is monitored by the many hardware and logical sensors present in phones nowadays which is then provided to the apps. Clearly, such information is very sensitive and has grave privacy implications if it falls in the wrong hands.

To avoid such personal data misuse, most mobile platforms including Meego [1], Android [2], Windows Phone (WP7) [3], provide a ‘needs’-based access control model where access to data collected by a sensor is only given to an app after explicit authorization by the user. At a high level, the model works as follows:

- Apps declare the list of sensors to which they need access (to provide their functionality) in their XML based Manifest files.
- During installation, the Manifest file is read and the list of required sensors’ access is presented to the user in a user-friendly language.
- The app is allowed to be installed only if the user says ‘Yes’. After installation, the mobile OS provides the needed access control to ensure that the app is only allowed access to those sensors as declared in its Manifest file.

While this acts as a deterrent, studies [4] have shown that such a model is not sufficient by itself. Many apps (including some of the most popular ones) have been observed to misuse the install-time access given to them at run-time. E.g. while a weather app requires legitimate access to the user’s location, install-time verification as above would be unable the prevent the app from retrieving the user’s location every few seconds and feeding it to an external server. A more fine grained run-time access control model [5] is not the solution either as they are practically infeasible from a performance perspective. The fundamental aspect missing here is the ability to express and control run-time characteristics of the app, such as the frequency of access.

Our first contribution is thus to propose a more expressive policy language which is able to capture the following (in addition to the specified list of sensors):

- Frequency of access.
- Sequence of access: It might not be acceptable by the user for an app to access a combination of sensors, but it might be OK if they are accessed in a specific sequence.
- Restricting the location/time of access: E.g. work related apps should only be accessing sensors’ data when the user is in office or only during office hours.

While such an expressive policy language enables the user to better control an app’s access to his personal data, it also makes the task of an average user more difficult in setting and analyzing the consequences of his privacy settings. In this paper, we propose a privacy add-on that at install-time compares the app’s requirements vs. user’s privacy settings, outlining the points of conflict as well as the different ways

Abstract—The ever increasing popularity of apps stems from their ability to provide highly customized services for the user. The flip side is that to provide such customized services, apps need access to very sensitive personal user information. This has led to a lot of rogue apps that e.g. pass personal information to 3rd party Ad servers in the background. Studies have shown that current app vetting processes which are mainly restricted to install time verification mechanisms are incapable of detecting and preventing such attacks. We argue that the missing fundamental aspect here is the inability to capture and control run-time characteristics of apps, e.g. we need to know not only the list of sensors that need to be accessed by an app but also their frequency of access. This leads to the need for an expressive policy language that in addition to the list of sensors, also allows specifying when, where and how frequently can they be accessed.

An expressive policy language has the disadvantage of making the task of an average user more difficult in setting and analyzing the consequences of his privacy settings. Further, privacy policies evolve over time. Over time, users are likely to change their privacy settings, as a response to a recently discovered vulnerability, or to be able to install that “much desired” app, etc. Such a policy change affects both already installed (may no longer be compliant) and previously rejected apps (may be compliant now). In this paper, we propose an integrated privacy add-on that (i) compares the apps profiles vs. user’s privacy settings, outlining the points of conflict as well as the different ways in which they can be resolved. And (ii) provides efficient change management with respect to any changes in user privacy settings.

Index Terms—Privacy policies; Conflict detection; Policy evolution; Smartphone apps/services;

I. INTRODUCTION

Apps (application for smartphones) are the lifeline of today’s smartphones. Their immense popularity is evident from the thousands of apps available in Nokia’s Ovi Store, Apple’s AppStore, Google’s Android Marketplace, Windows Marketplace Hub, etc. The reason apps are so popular and useful is because they provide services highly customized to different aspects of our life, from recommending location based places of interest to monitoring our health. The downside is of course that to do so apps collect a lot of personal information about the user. This personal information is monitored by the many hardware and logical sensors present in phones nowadays which is then provided to the apps. Clearly, such information is very sensitive and has grave privacy implications if it falls in the wrong hands.

To avoid such personal data misuse, most mobile platforms including Meego [1], Android [2], Windows Phone (WP7) [3], provide a ‘needs’-based access control model where access to data collected by a sensor is only given to an app after explicit authorization by the user. At a high level, the model works as follows:

- Apps declare the list of sensors to which they need access (to provide their functionality) in their XML based Manifest files.
- During installation, the Manifest file is read and the list of required sensors’ access is presented to the user in a user-friendly language.
- The app is allowed to be installed only if the user says ‘Yes’. After installation, the mobile OS provides the needed access control to ensure that the app is only allowed access to those sensors as declared in its Manifest file.

While this acts as a deterrent, studies [4] have shown that such a model is not sufficient by itself. Many apps (including some of the most popular ones) have been observed to misuse the install-time access given to them at run-time. E.g. while a weather app requires legitimate access to the user’s location, install-time verification as above would be unable the prevent the app from retrieving the user’s location every few seconds and feeding it to an external server. A more fine grained run-time access control model [5] is not the solution either as they are practically infeasible from a performance perspective. The fundamental aspect missing here is the ability to express and control run-time characteristics of the app, such as the frequency of access.

Our first contribution is thus to propose a more expressive policy language which is able to capture the following (in addition to the specified list of sensors):

- Frequency of access.
- Sequence of access: It might not be acceptable by the user for an app to access a combination of sensors, but it might be OK if they are accessed in a specific sequence.
- Restricting the location/time of access: E.g. work related apps should only be accessing sensors’ data when the user is in office or only during office hours.

While such an expressive policy language enables the user to better control an app’s access to his personal data, it also makes the task of an average user more difficult in setting and analyzing the consequences of his privacy settings. In this paper, we propose a privacy add-on that at install-time compares the app’s requirements vs. user’s privacy settings, outlining the points of conflict as well as the different ways
The other novel aspect considered in the paper is the long term evolution of the privacy policies. Over time, users are likely to change their privacy settings, as a result of a friend’s recommendation, or security advisory, or to adapt the settings to be able to install that “much needed” app, etc. If the change is user-initiated, then the user needs to be informed of the effect the policy change will have on already installed apps, as it is possible that the sensor requirements of some installed apps will cease to be compliant with respect to the modified privacy policy. On the other hand, some of the apps which were rejected earlier due to privacy policy conflicts may be compliant now. An analogous scenario arises when an app were rejected earlier due to privacy policy conflicts may be revalidated against the current privacy settings of the user. This assumes the existence of a location based ontology which can be broad as a list of cities, countries, etc. to more semantic ones e.g. office, home, restaurant, etc.

To capture the sequence in which the sensor can be accessed to retrieve their readings, we use a fragment of Metric First Order Temporal Logic (MFOTL) [8] to define the grammar for a sequence of sensor invocations as follows:

$$\phi_s ::= \text{sens}(s_i, f, sT, eT, r, L) \mid (\phi_s \land \phi_s) \mid (\lnot \phi_s) \mid (\phi_s \circ I \phi_s) \mid (\phi_s \bullet I \phi_s) \mid (\phi_s \cup I \phi_s)$$

where $s_i \in S$ refers to the set of available sensors and $I$ denotes the time interval $[a, b]$ with $a \leq b$ in a gradually increasing finite time domain. We use standard conventions to omit parentheses, e.g., temporal operators bind weaker than boolean connectives. Furthermore, the classical unary temporal operators are defined as follows: $\lnot \theta := \text{true} \ S \theta$ (once), $\lozenge \theta := \text{true} \ T \theta$ (always in the past), $\Diamond \theta := \text{true} \ U \theta$ (eventually), and $\Box \theta := \lnot \lozenge 1 \theta$ (always), where $I \in \mathbb{I}$. For a detailed description of the semantics of the different temporal operators, the interested reader is referred to [8].

For simplicity, we only consider the unary temporal operator $\lozenge$ in the sequel. Even then the grammar defined above is quite expressive, as it not only allows to specify a sequence of sensor invocations but also the time interval between subsequent invocations. E.g., the sequential invocation of three sensors $s_1, s_2, s_3, s_4$ in the next 30 minutes after $s_1$’s invocation, followed by $s_3$ within the next 40 minutes can be specified as follows:

$$\phi_s ::= \text{sens}(s_1, f_1, \cdots) \land \lozenge [0, 30] \text{sens}(s_2, f_2, \cdots) \land \lozenge [30, 70] \text{sens}(s_3, f_3, \cdots)$$

Note that the sequence grammar can also be used to denote simultaneous invocations (setting the time intervals to $[0, 0]$) or simply that a group of sensors need to be invoked in an ad-hoc manner (using only logical AND without time intervals). So a sequence specification is also sometimes referred to as a sensor invocation pattern in the paper depending on the context.

### III. Privacy Policy Management

#### A. Conflict Detection

**User privacy profile:** Based on the policy language defined in Section II, the privacy profile $u_P$ of user $u$ can be defined as the set of of permitted sensor invocation sequences i.e.

$$u_P ::= (\phi_{u_1} \lor \phi_{u_2} \lor \cdots)$$

Note that there can be more than one sensor invocation sequence allowed by a user for the same group of sensors, so the need for multiple allowed sequences.

**App profile:** On the same lines, the app profile $a_P$ of an app can be defined as a list of the sensor invocation sequences it needs to fulfill its functionality.

$$a_P ::= (\phi_{a_1} \land \phi_{a_2} \land \cdots)$$

Given $u_P$ and $a_P$ then, the following algorithm can be used to detect conflicts while installing app $a$ on user $u$’s phone. With an expressive policy language, there can clearly be more than one possible reason for conflict. Here we take

- Evaluate the effect of a privacy policy change on already installed apps.
- Retain a history of rejected apps (with reason for rejection), recommending those that become compliant in the event of a policy change.
- Re-validate an installed app with respect to any changes in its sensor access requirements over time.

To the best of our knowledge, change management in literature has only focused on large scale software project management [6], [7] till now, and their usage with respect to privacy policies is a completely novel aspect of this work.

The rest of the paper is organized as follows: In the next section, we outline our privacy policy language. Algorithms for conflict detection, resolution, and change management are then provided in Section III. Section IV gives implementation details of our prototype on the Nokia N900. Finally, Section V concludes the paper with directions for future work.

### II. Privacy Policy Language

In this section, we outline the privacy policy language used by the both the app providers to specify their sensor access requirements and for users to specify their privacy profiles. We give a logic based policy language in the sequel.

The sensors in the phone, e.g. Accelerometer, GPS, etc. are represented as predicates $\text{sens}(s_i, f, sT, eT, r, L)$ with the following parameters:

- $s_i$ is the sensor identifier.
- $f$: the frequency of invocation. The value can also be $-1$ indicating a more ad-hoc needs based invocation pattern.
- $sT, eT, r$: The start $sT$ and end times $eT$ during which the sensors would be invoked (can be invoked from the user’s perspective). The flag $r$ denotes if the start and end times are recurring, e.g. hourly, daily, monthly, etc.
- $L$: set of locations where the sensor would be invoked. This assumes the existence of a location based ontology
a hierarchical approach, first trying to detect any conflicts with respect to list of required sensors, followed by the sequences in which the sensors can be invoked, followed by each sequence interval constraints, and finally parameter level conflicts corresponding to each sensor invocation with respect to when, where and with what frequency can the sensor be accessed. The hierarchy is illustrated in Fig. 1.

Before outlining the algorithm, we need to define the ‘subsequence’ relation:

**Definition 1 (Sub/Supersequence):** Given sequences $\phi_f_1$ and $\phi_f_2$, let $\phi_{f_1}'$ and $\phi_{f_2}'$ denote their corresponding transformed sequences retaining only the sensor identifier $s_i$ as parameter of the sens predicates and ignoring the temporal constraints. Then $\phi_{f_1}$ is a subsequence of $\phi_{f_2}$ iff $\phi_{f_1}'$ is a fragment of $\phi_{f_2}'$, i.e. $\phi_{f_2}' := \text{sens}(s_i) \land \cdots \land \phi_{f_1}' \land \cdots$. Analogously, $\phi_{f_2}$ is a supersequence of $\phi_{f_1}$.

**Conflict Detection Algorithm**

1) **Sequence:** For each $\phi_{a_i}$ in $a_P$, check if $\phi_{a_i}$ is a subsequence of at least one sequence $\phi_{a_j}$ in $u_P$. If not, then there is a conflict. Otherwise, consider the subsequence pair $\phi_{a_i}$-$\phi_{a_j}$ for the remaining steps. [Note that sensor mismatches (a required sensor is not allowed by the user in any sequence) are implicitly captured by the sequence definition. Further, there can be more than one potential supersequence at this stage. So if a conflict is detected in the latter steps then the process needs to be repeated with the next potential supersequence, and so on till either a match is found or all potential supersequences have been exhausted.]

2) **Sequence interval:** For each $\text{sens}(s_i, \cdots)$ predicate in $\phi_{a_i}$ and the corresponding $\text{sens}(s_j = s_i, \cdots)$ in $\phi_{u_i}$, let $t_{a_i} = \lceil a, b \rceil$ and $t_{u_i} = \lceil c, d \rceil$ denote their respective temporal quantifiers. Given this, there is a conflict if $(a < c)$ or $(b > d)$.

3) **Frequency:** For each $\text{sens}(s_i, f_i, \cdots)$ predicate in $\phi_{a_i}$ and the corresponding $\text{sens}(s_j = s_i, f_j, \cdots)$ in $\phi_{u_i}$, there is a conflict if $f_i > f_j$.

4) **Timing:** For each $\text{sens}(s_i, f_i, sT_i, eT_i, r_i, \mathcal{L}_i)$ predicate in $\phi_{a_i}$ and the corresponding $\text{sens}(s_j = s_i, f_j, sT_j, eT_j, r_j, \mathcal{L}_j)$ in $\phi_{u_i}$, there is a conflict if $r_i > r_j$, or if $r_i \leq r_j$ but $sT_i < sT_j$ or $eT_i > eT_j$. Comparing $r_i$ and $r_j$ assumes the existence of an ordered recurrence pattern ontology, e.g. ‘daily’ < ‘monthly’.

5) **Location:** For each $\text{sens}(s_i, \cdots, \mathcal{L}_i)$ predicate in $\phi_{a_i}$ and the corresponding $\text{sens}(s_j = s_i, \cdots, \mathcal{L}_j)$ in $\phi_{u_i}$, there is a conflict if $\mathcal{L}_i \notin \mathcal{L}_j$.

**B. Conflict Resolution**

When a conflict is detected between an app’s sensor invocation requirements and user’s privacy policy, the currently used approach in most mobile platforms is simply to reject the app. We argue that this is insufficient in the long run. Clearly, the user is interested in installing the app, so a more ‘user-friendly’ privacy management tool would not only explain the conflict to the user but also outline how to resolve those conflicts. In the end, it is of course always the user’s prerogative to accept or reject any of the proposed resolutions.

For each type of conflict, we give the conflict explanation and a possible resolution which can e.g. be used as a basis to explain the conflict and proposed resolution to the user. Other resolution strategies can be analogously devised.

**Resolution Strategy**

1) **Sequence:** For a $\phi_{a_i}$ in $a_P$, there does not exist a supersequence in $u_P$ of $\phi_{a_j}$. Determine the largest set of sensors $f$ such that $s \in S(\phi_{a_i})$ and $s \in S(\phi_{a_j})$, where $S(\phi)$ refers to the set of sensors $\{s_i|\text{sens}(s_i, \cdots) in \phi\}$.
   - $|s| = |S(\phi_{a_i})|$: [Explanation] A supersequence $\phi_{a_j}$ containing the required set of sensors exists, but the invocation pattern is different. [Resolution] Adapt the sequence $\phi_{a_i}$ accordingly.
   - $|s| < |S(\phi_{a_i})|$: [Explanation] The subset of sensors $s' = S(\phi_{a_i}) \setminus s$ is not permitted by the user. [Resolution] Add the subset of sensors $s'$ and adapt the sequence $\phi_{a_j}$ accordingly.
   - $|s| = 0$: [Explanation] Access requested for a completely novel set of sensors. [Resolution] Add the requested sequence $\phi_{a_i}$ to the user privacy policy $u_P$ (i.e. allow the sensor invocation pattern $\phi_{a_i}$).

2) **Sequence interval:** [Explanation] For a $\phi_{a_i}$ in $a_P$, there exists a supersequence $\phi_{a_j}$ in $u_P$ but the intervals among subsequent sensor invocations do not match. [Resolution] Let $t_{a_i} = \lceil a, b \rceil$ and $t_{u_i} = \lceil c, d \rceil$ denote the mismatched intervals and $b > d$ ($a < c$). Then a possible resolution would be to increase the upper limit $d$ (decrease the lower limit $a$).

Note that increasing $b$ implicitly increases the ‘absolute’ time intervals of all following (preceding) sensor predicates in $\phi_{a_j}$. So such an increase (decrease) would need to be validated in the scope of the whole sequence $\phi_{a_j}$ as well.

3) **Frequency, Location:** [Explanation] We focus on parameter mismatches between sensor predicates
\(sens(s_i, f_i, \ldots, L_i)\) and \(sens(s_j = s_i, f_j > f_i, \ldots, L_j \supset L_i)\) in \(\phi_u\), and its supersequence \(\phi_{u_j}\), respectively. \[\text{Resolution}\] W.r.t. frequency, allow increased frequency of accessing sensor \(s_i\), i.e. decrease \(f_j\). And with respect to the set of permitted locations, add the subset of locations \(L_i \setminus L_j\) to \(L_j\).

4) **Timing:** \[\text{Explanation}\] Conflicts with respect to the timings when a sensor can be accessed to retrieve its data. \[\text{Resolution}\] Conflicts in recurrence (\(r\)) and timing (\(sT, eT\)) patterns can be resolved analogous to the approaches discussed earlier for frequency and sequence interval conflicts respectively.

We have presented the resolution strategies from the user’s point of view, i.e. highlighting the modifications required in the user’s privacy policy to resolve a conflict. The other direction of providing conflict details of a failed app installation to the app providers might be useful as well. While it is impractical to assume that app providers would adapt the sensor requirements of their apps (possibly modifying their functionality as well) for a single user, the collective feedback would hopefully allow them to better shape their future offerings.

C. Policy Evolution

In the previous section, we saw an instance of policy change as a result of conflict resolution (policy conflicts while installing an app may necessitate a change in the user’s privacy settings). Changes in policies may also be triggered on the recommendation of a friend or security advisory to protect the phone against a recently discovered vulnerability. Whatever the reason might be, it is inevitable that policies will evolve over time. Thus any privacy management tool should be able to adapt according to the privacy policy evolution.

While the effect of a user policy change on already installed apps is expected, we argue that it is recommended to analyze their effect on previously rejected apps as well. We envision the following main functionalities as part of the change management functionality of the policy management tool:

- **Installed apps:** For any change in the user’s privacy settings to be effective, the sensor requirements of already installed apps need to be re-evaluated in light of the change, to assess if they are still compatible with the updated privacy policy.
- **Rejected apps:** For previously rejected apps, it also makes sense to re-evaluate their compatibility in the event of a policy change, as the change might have resolved the policy conflict which had caused the app to be rejected in the first place.

In the sequel, we show how the above change management functionalities can be achieved in an efficient manner.

To enable change management, we maintain two logs: (i) Installed apps with their profiles, and (ii) Rejected apps categorized by their conflict types, e.g. sensor mismatch, sequence (interval) mismatch, parameter mismatches, etc. Recall that there might be more than one possible type of conflict between an app and user privacy profile at a time.

The changes in user privacy policy \(u_P\), as with conflict detection and resolution, can also be of different types. Rather than checking all installed and rejected app profiles for each change in \(u_P\), we take the following more efficient approach. We first classify any change in \(u_P\) as ‘relaxing’, ‘restricting’, or ‘general’.

- **Sequence:**
  - **Relax:** Adding a completely new sequence to \(u_P\) is clearly a relaxation of the user privacy policy \(u_P\). In addition, a sequence \(\phi_{u_j}\) in \(u_P\) is said to be relaxed if access to a new sensor is allowed with the corresponding sensor predicate added at the beginning or end of sequence \(\phi_{u_j}\).
  - **Restrict:** Both deleting an entire sequence \(\phi_{u_j}\) from \(u_P\) and deleting a sensor predicate \(sens(s, \ldots)\) from \(\phi_{u_j}\) (revoking access to sensor \(s\)) have a restricting effect on \(u_P\).
  - **General:** Adding a sensor predicate \(sens(s, \ldots)\) to \(\phi_{u_j}\), other than at the beginning or end of \(\phi_{u_j}\). It is difficult to characterize such an addition as either relaxing or restricting as it has traits of both: adding a sensor predicate \(sens(s, \ldots)\) implies relaxing \(u_P\) by allowing access to a new sensor \(s\), at the same time it also imposes additional sequencing restrictions with respect to invoking \(s\) in between the sensors preceding and following it in \(\phi_{u_j}\).

- **Sequence interval:**
  - **Relax:** For an interval \(t_u = \Diamond[a, b]\), relaxing implies increasing the upper limit \(b\) or decreasing the lower limit \(a\).
  - **Restrict:** On the same lines, increasing \(a\) or decreasing \(b\) has a restricting effect on \(t_u\).

- **Frequency:**
  - **Relax:** For a sensor predicate \(sens(s_j, f_j, \ldots)\) in \(\phi_{u_j}\), relaxing parameter \(f_j\) implies allowing sensor \(s_j\) to be accessed more frequently, i.e. decreasing \(f_j\).
  - **Restrict:** Increasing \(f_j\).

- **Location:**
  - **Relax:** For a sensor predicate \(sens(s_j, \ldots, L_j)\) in \(\phi_{u_j}\), relaxing the set of permitted locations \(L_j\) implies adding a novel location \(l_k \notin L_j\) to \(L_j\).
  - **Restrict:** Deleting a location \(l_k \in L_j\) from \(L_j\).

- **Timing:**
  - **Relax:** For a sensor predicate \(sens(s_j, f_j, sT_j, eT_j, r_j, \ldots)\) in \(\phi_{u_j}\), lowering the recurrence pattern \(r_j\) (from ‘monthly’ to ‘daily’), and increasing the allowed end-time \(eT_j\) (decreasing the allowed start-time \(sT_j\)) are considered as relaxing the timing parameter constrains with respect to accessing sensor \(s_j\).
  - **Restrict:** Upping the recurrence pattern \(r_j\), and decreasing the allowed end-time \(eT_j\) (or increasing the allowed start-time \(sT_j\)).
  - **General:** The following change cannot be characterized as strictly relaxing or restricting: lowering...
the recurrence pattern \( r_j \), but decreasing the allowed end-time \( eT_j \) (or increasing the allowed start-time \( sT_j \)).

Classifying any changes in the user privacy policy \( u_P \) as ‘relax’, ‘restrict’, ‘general’ enables optimizing the change management process as follows:

- **Relax**: For changes which have a relaxing effect on \( u_P \), only previously rejected apps need to be re-evaluated for compliance. All installed apps remain compliant.
- **Restrict**: For restricting type of changes, only installed apps need to be re-evaluated for compliance. Previously rejected apps will remain non-compliant.
- **General**: Both installed and rejected apps need to be re-evaluated for non-compliance and compliance respectively.

Further optimization techniques can also be used in selecting the installed/rejected apps to be re-evaluated on a policy change. E.g. a policy change with respect to sequence \( \phi_{u_i} \) in \( u_P \) would only affect those apps \( a \) whose profiles \( a_P \) contain a subsequence \( \phi_{a_k} \) of (super)sequence \( \phi_{u_j} \).

### IV. IMPLEMENTATION

We have implemented the privacy management algorithms described in this paper as an add-on to the Application Manager on a Nokia N900 phone running the Maemo 5 operating system. The underlying policy evaluation engine is developed in Prolog (SWI-Prolog). Prolog supports relations which are represented by facts and rules. Facts and rules are stored in an internal knowledge database. Users can run queries over these relations and the Prolog interpreter evaluates them to show results that satisfy the given goal. In our system, policies in the (current) user privacy profile are maintained as facts in the Prolog database, with the app profiles given as queries to the Prolog interpreter. We give a sample use-case below illustrating the implementation output.

Let the initial user privacy profile \( u_p \) be to allow apps to only access the Wi-fi sensor with a frequency: 5 times/sec. The following fact is then added to the internal knowledgebase maintained by the tool:

\[
\text{sens(wifi, 5, \_ - \_ - \_)}
\]

Apps with extended privacy profiles are simulated by stating their sensor usage in the “Descriptions” section of the published apps. The sample apps are available published in the Extras-devel\(^1\) Maemo apps repository. Let us consider an app \( A \) requiring access to the GPS and Wi-fi sensors with the following usage patterns:

\[
\phi_{a_1} = \text{sens(wifi, 10, \_ - \_ - \_ - \_)} \cup \text{sens(gps, 1, \_ - \_ - \_)}
\]

Giving the predicates in \( \phi_{a_1} \) as a query to our logic engine leads to the following output (conflict descriptions with their possible resolutions):

**Output:**

\[^1\text{http://wiki.maemo.org/Extras-devel}\]

---

In spite of the possible resolutions, let the user reject the app \( A \). Given this, the rejected app \( A \)’s profile is added to the Rejected apps log (both under the categories ‘sensor mismatches’ with respect to GPS and ‘parameter mismatches’ with respect to frequency).

Now, let the user modify his privacy profile by increasing the allowed frequency of Wi-fi to 15 times/sec. As this is a ‘relaxation’ of the privacy profile, the already installed apps remain valid and are not affected by this change. To evaluate its effect on previously rejected apps, the modified rule is now given as a Prolog query with the Rejected apps log acting as the facts knowledgebase.

**Output:**

\[^2\text{http://projects.developer.nokia.com/MonPoly}\]

---

### V. CONCLUSION

In this paper, we considered the problem of regulating access by 3\(^rd\) party apps to sensors available in the phone. We advocated the need for a more expressive privacy policy language including parameters such as the frequency of access, location, timing, etc. With both user privacy settings and app sensor usage patterns specified in such an expressive language, we gave efficient algorithms to detect conflicts and outline possible resolution strategies. As policies evolve over time, a change management framework is also provided to deal with the modifications in a consistent fashion, (re-)checking the validity of both already installed apps and those that were rejected earlier. Finally, prototype details of an implementation on the Nokia N900 are given to show the practical feasibility of the proposed concepts.

The current implementation lacks the ability to handle temporal constraints as allowed by the policy language. To overcome this, work is ongoing to replace the Prolog based underlying logic engine with the MonPoly MFOTL engine [9]. MonPoly is available freely\(^2\) under the open-source LGPL 2.1 license. Future work also includes plans to extend the current formal policy language to XACML [10], allowing the specification and algorithms to be implemented in a more generic fashion portable across mobile platforms.
ACKNOWLEDGEMENT

We would like to thanks the anonymous referees for their helpful suggestions that helped to improve the work in this paper considerably.

REFERENCES


