Automatic Mediation of Privacy-Sensitive Resource Access in Smartphone Applications

Benjamin Livshits and Jaeyeon Jung
Microsoft Research

Abstract

Mobile app development best practices suggest that developers obtain opt-in consent from users prior to accessing potentially sensitive information on the phone. We study challenges that mobile application developers have with meeting such requirements, and highlight the promise of using new automated, static analysis-based solutions that identify and insert missing prompts in order to guard otherwise unprotected resource accesses. We find evidence that third-party libraries, incorporated by developers across the mobile industry, may access privacy-sensitive resources without seeking consent or even *against* the user's choice. Based on insights from real examples, we develop the theoretical underpinning of the problem of mediating resource accesses in mobile applications. We design and implement a graph-theoretic algorithm to place mediation prompts that protect every resource access, while avoiding repetitive prompting and prompting in background tasks or third-party libraries.

We demonstrate the viability of our approach by analyzing 100 apps, averaging 7.3 MB in size and consisting of dozens of DLLs. Our approach scales well: once an app is represented in the form of a graph, the remaining static analysis takes under a second on average. Overall, our strategy succeeds in about 95% of all unique cases.

1 Introduction

Privacy on smartphones is far from being a theoretical issue: a popular iOS application, Path, had been found to upload the entire address book of an iPhone user by default; similarly, a number of high-profile incidents [1–3] show negative consequences for mobile applications that surreptitiously collected privacy-sensitive information about users without explicit consent. Furthermore, a recent survey of 714 cell phone users shows that 30% of the respondents had uninstalled an application because they discovered that the application in question was collecting personal information they did not wish to share [20].

Runtime *consent dialogs* (sometimes called runtime permission prompts) are commonly used by mobile applications to obtain a user's explicit consent *prior* to accessing privacy-sensitive data. However, mobile operating systems differ in terms of their approach to raising these consent dialogs. iOS implements OS-level consent dialogs which are raised when accessing GPS location, contacts stored on the phone, as well as a few other key resources. These dialog boxes are far from being "no-ops" for the user: A recent study of hundreds of iPhone users shows that 85% of them exercised this control to deny at least one application from accessing location data [13]. However, in the absence of OS-level support, application developers can individually implement opt-in consent dialogs for enhancing the overall privacy for end-users.

This paper focuses on a number of technical challenges that arise when mobile application developers determine the right place to insert runtime prompts within an application. First, minimizing the runtime *frequency* of consent dialogs is important, as repetitive prompts tend to habituate users to blindly accept the terms [7]. However, to protect user privacy, every single attempt to access sensitive information should be guarded with a prompt. Second, apps should provide *just-in-time* prompts in order for it to make sense to the user within the application context. If prompts are placed early, e.g., at *install time*, users may forget about granted permissions, leading to unpleasant surprises because of data access performed by the app, especially when it runs in the background [21].

The aim of this paper is to formalize the problem of placing runtime consent dialogs within a mo-

bile application, and to propose a solution for automatic and correct prompt placement. We try to both 1) find missing prompts and 2) propose a valid prompt placement when prompts are missing.

1.1 Analysis Design Philosophy

While it is possible to use dynamic analysis to observe missing prompts at runtime, this approach is fraught with significant challenges. The traditional challenge is low path coverage, which can be alleviated with path exploration techniques such as symbolic execution, but never completely fixed. Other, more technical, challenges related to running UI-based mobile apps automatically also remain.

Because we aim to provide a technique that would err on the side of safety, we do not believe runtime analysis is suitable. To this end, we propose a new scalable static analysis algorithm to automatically find places for inserting prompts if they are missing. Our solution scales well with application size and does not require any changes to the underlying operating system.

Given the inherent nature of static analysis techniques and the complexity of both the applications and the execution environment, our tool may produce false positives. However, at the worst, these false positives will result in double-prompts that occur at most once per application. We believe this to be a considerable improvement over the current error-prone practice of manual prompt placement. Our approach in this paper may not be fully sound due to issues such as reflection (see Section 4); however, our goal is be as sound as possible. Our evaluation in Section 6 does not reveal any false negatives.

Finally, note that our target is benign, but potentially buggy non-obfuscated apps. If the app writer tries to either obfuscate their code or take advantage of features that are not treated conservatively (such as reflection) to hide control flow, the precision and soundness of our analysis will suffer. Luckily, the presence of obfuscation is relatively easy to detect [22].

1.2 Contributions

Our contributions are three-fold:

- Using a set of .NET WP (Windows Phone) applications, we study how existing applications implement resource access prompts. We note that some advertising libraries access location data without a prompt.
- We propose a two-prong static analysis algorithm for correct resource access prompt placement. We first attempt to use a fast, dominator-based placement technique. If that fails, we resort to a slower but more exhaustive backward search.
- We evaluate our approach to both locating missing prompts and placing them when they are missing on 100 apps. Overall, our two-prong strategy of dominator-based and backward placement succeeds in about 95% of all unique cases. Our analyses run in seconds, making it possible to run them as part of the app submission process.

Our analysis reveals that some application developers fail to show the proper set of prompts, showing the difficulty and ineffectiveness of manual placement. Frequently, the issue that exacerbates this situation is that resource access takes place within third-party libraries shipped as bytecode, making them more difficult to reason about largely placing them outside developer's control.

1.3 Paper Organization

The rest of this paper is organized as follows. We discuss case studies of real applications and challenges associated with proper prompt placement in Section 2. We then formulate the problem and provide much of the insight for our proposed solution in Section 3. We discuss the implementation of the algorithms in Section 4. Results from an experimental study are described in Section 5 and further discussed in Section 6. We summarize related work in Section 7 and conclude in Section 8.

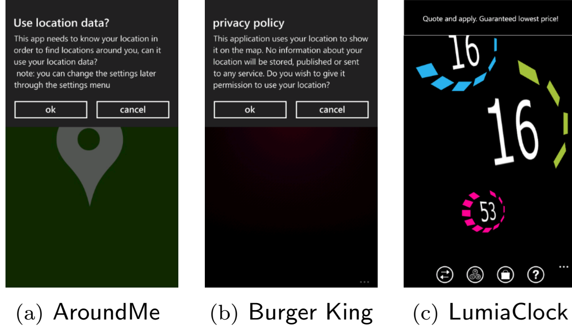
2 Background

We first provide three motivating case studies in Section 2.1 and then provide intuition for the complexity of the problem in Section 2.2.

2.1 Motivating Case Studies

We begin by discussing several interesting real-world examples, which illustrate how existing WP apps mediate access to location data. One of the ways in which the WP SDK exposes location access API to applications is through the `GeoCoordinateWatcher` class in the `System.Device.Location` namespace. Prompts are created with a call to `MessageBox.Show`, with the text of the prompt provided by the developer.

Figure 1 shows screen-shots of three applications — AroundMe, Burger King (inoffiziell), LumiaClock — immediately before these applications in-



(a) AroundMe (b) Burger King (c) LumiaClock

Figure 1: Screen-shots of three examined applications. The first two applications display a location prompt prior to invoking location APIs. The third application never shows a location prompt; the screen-shot was captured when we detected the first time that a location API was invoked by the application.

App	Resource accesses	APIs used	Libraries
AroundMe	2	`TryStart`, `get_Position`	`AroundMe.dll`
Burger King	5	`Start`, `get_Position`	`BurgerKing.dll`, `GART.dll`
LumiaClock	2	`Start`, `get_Position`	`SOMAWP7.dll`

Figure 2: Location accesses found in three apps.

voke location access API. We picked these three apps from the WP Store, filtering for apps that use GPS location data. Each application consists of a set of DLLs and resources. We have disassembled the applications and inspected the code to find instances of location API invocations. Figure 2 shows (1) the number of location access points observed in each of the three applications; (2) which location API is used; and (3) which libraries call the location API.

As shown in Figure 2, location access happens both in application code and in third-party libraries. For instance, `GART.dll` is a library that provides augmented reality features and `SOMAWP7.dll` is a library that provides advertising to WP applications. Not surprisingly, the use of location data by third-party libraries complicates access mediation, as third-party libraries often come as a black box to application developers. The following in-depth analysis illustrates the issue.

Case 1 (proper protection): Location accesses are contained only in the application code and properly mediated by a runtime consent dialog. The code snippet in Figure 3(a) is from the `AroundMe` application. As shown in the code below, this application

```
public static bool AroundMe.App.CheckOptin() {
 if (((Option)Enum.Parse(typeof(Option),Config.GetSetting
  (SettingConstants.UseMyLocation),true))==Option.Yes) {
  return GetCurrentCoordinates();
 }
 if (MessageBox.Show("This app needs ...",
                             "Use location data?",
  MessageBoxButton.OKCancel)==MessageBoxResult.OK) {
  Config.UpdateSetting(new KeyValuePair<string,string>
  (SettingConstants.UseMyLocation,Option.Yes.ToString()));
  return GetCurrentCoordinates();
 }
 ...
}
```

(a) Illustration for Case 1

```
public BurgerKing.View.MapPage() {
 this.InitializeComponent();
 base.DataContext = new MapViewModel();
 this.BuildApplicationBar();
 if (AppSettings.Current.UseLocationService){
  this.watcher = new GeoCoordinateWatcher();
 }
 ..
}

protected virtual void GART.Controls.ARDisplay.
  OnLocationEnabledChanged(
          DependencyPropertyChangedEventArgs e)
{
 if (this.servicesRunning) {
  if (this.LocationEnabled) {
   this.StartLocation();
  }
  else {
   this.StopLocation();
  }
 }
}
```

(b) Illustration for Case 2

```
public SomaAd()
{
...
 this._locationUseOK = true;
...
 if (this._locationUseOK) {
  this.watcher = new GeoCoordinateWatcher
   (GeoPositionAccuracy.Default);
  this.watcher.MovementThreshold = 20.0;
  this.watcher.StatusChanged += new EventHandler
   <GeoPositionStatusChangedEventArgs>(
   this.watcher_StatusChanged);
  this.watcher.Start();
 }
}
```

(c) Illustration for Case 3

Figure 3: Illustrative cases for Section 2.1.

invokes `GetCurrentCoordinates`() only after the user clicks the `OK` button as shown in Figure 1.

Case 2 (partial protection): Location accesses are spread across application and third-party code and only accesses by application code are protected by runtime consent dialog. The code snippet in Figure 3(b) is from the BurgerKing application. The consent dialog shown in Figure 1 only affects `AppSettings.Current.UserLocationService` and leaves `GART.Controls.ARDisplay.StartLocation`() unprotected. Using network packet inspection, we con-

```
while(P){
    l1 = getLocation();
}
```

(a) original

```
prompt();
while(P){
    l1 = getLocation();
}
```

(b) static prompt

```
while(P){
    if(not-yet-prompted-for-location){
        prompt();
    }
    l1 = getLocation();
}
```

(c) dynamic check

Figure 4: Resource access in a loop.

firmed that the application accesses and transmits location using the `GART` component *even* when the `Cancel` button is clicked.

Case 3 (no protection): Location accesses are only present in third-party code and the application provides no consent dialogs. The following code snippet is from the LumiaClock application. The application has no location features. Although the third-party code `SomaAd` exposes a flag to protect location access, the application appears unaware of it. Moreover, the `SomaAd` component enables the flag, `_locationUseOK` by default, as shown in Figure 3(c).

Summary: In summary, the case studies above demonstrate that properly protecting location access is challenging because multiple components, including third-party libraries, are involved in accessing sensitive resources. The current practice often fails in providing adequate privacy protection, as some applications do not honor the user's choice (as shown in case 2) or do not obtain the user's consent prior to acquiring privacy-sensitive information.

2.2 Challenges

Next, we dive into the properties that we want to ensure, while deciding where to place missing prompts via static analysis. Naïvely, one might suspect that prompt placement is a fairly trivial task, reducing to (1) finding resource access points and (2) inserting prompts right in front of them. In reality, situation is considerably more complex. In this section, we systematically investigate the challenges we need to overcome in order to provide a satisfactory solution.

1) Avoiding double-prompts: We need to avoid prompting the user for access to resource R more than once on a given execution path. This is a harder problem that it might initially seem; indeed, consider the following code:

```
if(P) l1 = getLocation();
l2 = getLocation();
```

There are two location access points and two ways to avoid duplicate prompts. One is to introduce a boolean flag to keep track of whether we have prompted for the location already:

```
flag = true;
if(P){
    prompt();
    flag = true;
    l1 = getLocation();
}
if(!flag){
    prompt();
    l2 = getLocation();
}
```

The disadvantage of this approach is that it requires introducing extra runtime instrumentation to perform this sort of bookkeeping. A fully static approach involves rewriting the original code by "folding" the second prompt into the `if`:

```
if(P){
    prompt();
    l1 = getLocation();
    l2 = getLocation();
}else{
    prompt();
    l2 = getLocation();
}
```

This approach has the advantage of not having to introduce extra bookkeeping code. The disadvantage is replication of the existing code across the branches of the `if`, which leads to extra code growth.

The problem of double-prompts can be exacerbated. Figure 4a illustrates the challenge of placing a prompt within a loop. Placing the prompt before the loop as in Figure 4b is not valid if the loop never executes. Placing the prompt within the loop body will lead to execution on every iteration. However, a simple dynamic check will ensure that the location prompt is not shown more than once (Figure 4c).

2) Sticky prompts: Applications frequently make user-granted permissions persistent and avoid duplicate prompts, by saving the prompt status to the app's isolated storage, as illustrated in Figure 5. Here the challenge comes in both recognizing existing "sticky" prompts in app code and in making inserted prompts sticky, as discussed in Section 4.3.

3) Avoiding weaker prompts: Suppose there are two resources r_1, r_2 such that r_2 is less sensitive than r_1. If an app has already prompted the user for

access to r_1, it should avoid prompting the user for access to resource r_2. For instance, if an app already has requested access to fine-grained location, there is no need to prompt for access to coarse-grained location. Note that in the current version of the WP operating system, there is no difference in capabilities between fine- and coarse-grained locations; both require the `ID_CAP_LOCATION` capability in the app manifest. However, in the future more fine-grained capabilities subsuming one another may evolve, as they have on Android. Moreover, it is still possible and perhaps even desirable to distinguish between fine- and coarse-grained locations when prompting at runtime, even though they are treated the same at installation time.

4) Avoiding prompts in background tasks: WP apps provide non-interactive background tasks. These are often used for polling remote servers and other tasks that do not require access to the user's screen beyond, perhaps, a live tile of the app. We cannot raise dialog boxes within background tasks. To properly determine where the prompts should be located, we should compute the call graph and determine what foreground code precedes the code within background tasks.

5) Avoiding prompts in libraries: Given that libraries are often shipped in the form of bytecode and are updated separately from the rest of the applications, we choose to avoid placing prompts in library code. This approach allows developers to examine prompt placement within their own code, and to alleviate the need to keep custom-modified versions of third-party libraries such as `SOMAWP7.dll`, which can make error reporting, debugging, and sharing libraries across apps a challenge.

```
if (MessageBox.Show(
    "This app needs to know your location
        in order to find locations
        around you, can it use your location data?
        note: you can change the settings later
        through the settings menu",
        "Use location data? ", 1) == 1)
{
    Config.UpdateSetting(
        new KeyValuePair<string, string>(
            SettingConstants.UseMyLocation,
            Option.Yes.ToString()));
    return
            GetCurrentCoordinates();
}
```

Figure 5: Sticky location prompt.

3 Overview

A recent spate of research efforts is centered around detecting undesirable *information flows*, i.e. sensitive data like contacts leaving the phone, usually via the network (e.g., [9, 10]). Reasoning about these kinds of leaks involves understanding inter-procedural data flow within the app and perhaps even across different apps. Data flow analysis of this kind is a known difficult problem which, despite a great deal of work on both the static and runtime sides has not yet found widespread practical deployment [24].

In the context of mobile apps, there is another aspect further complicating this problem. Even if there is in fact a perfect *mechanism* for precisely and efficiently tracking inter-procedural data flow, a viable *policy* is hard to come by. Indeed, how does a tool automatically distinguish between a Yelp app that shares GPS location information with a back-end server to obtain restaurant listings from (a potentially malicious) flashlight app that obtains the same GPS information and shares it with an ad server? Constructing a robust policy is not trivial. Our paper rather focuses on providing a method for assisting application developers in checking their apps against the currently accepted practice of obtaining consent prior to accessing potentially sensitive user data on the phone and in fixing problems before submitting apps to a marketplace. Note that our work in this paper is in the control flow, *not* data flow space; we want to reason about whether the acquisition points for sensitive content are well-protected. In this section we first formulate the problem of prompt placement and then discuss some approaches for computing a valid placement.

3.1 Graph Representation

As is typical in static analysis, it is helpful to represent the program in the form of a graph, to abstract away many unnecessary features of the original source or bytecode representation.

Since our goal is to reason about prompts "guarding" resource access points, we choose a representation similar to a control-flow graph. Because both prompts and resource accesses take the form of method calls, we find it convenient to augment the traditional notion of *basic blocks* to treat call sites specially. We use the term *enhanced basic block* to emphasize the difference in construction. An enhanced basic block is different from a basic block in that only the first and last of its instructions can be (method) calls. Consequently, call instructions exist

in a block of their own. (First and last instructions can also be jumps, just as in the case of regular basic blocks.)

Our representations also need to be inter-procedural: we need to be able to handle prompts that are located outside of the method in which the resource access takes place. This is especially necessary given that WP apps are written in .NET, where methods generally tend to be small. We therefore augment the control glow graph with call and return edges denoted as C below.

Definition 1 *A resource access prompt placement problem is defined as follows. Let $\mathcal{P} = \langle N, A, B, E, C, \mathcal{L} \rangle$ be a tuple with the following components:*

- *N: set of* enhanced *basic blocks in the program consisting of a sequence of instructions $N = n_1, n_2, \ldots n_k$. For simplicity, we assume that graph G has unique entry and exit nodes $N_{\mathbf{entry}}, N_{\mathbf{exit}} \in N$.*
- *$A \subset N$: set of resource access points;*
- *$B \subset N$: set of enhanced basic blocks located within background tasks and (third-party) libraries; we assume that $N_{\mathbf{entry}}$ and $N_{\mathbf{exit}}$ are outside background tasks and libraries;*
- *E: intra-procedural control flow edges;*
- *C: inter-procedural call and return edges.*
- *$\mathcal{L} = \langle R, \wedge \rangle$: the semi-lattice of access permissions with meet operator $\wedge$*[1]*.*

Intuitively, this representation is an expanded inter-procedural control flow graph $G = \langle N, E \cup C \rangle$.

[1]We assume that in the general case it is possible for permissions to subsume one another, like in the case of fine- and coarse-grained GPS locations, giving rise to a partial order, although we currently do not strictly need this kind of support in our implementation.

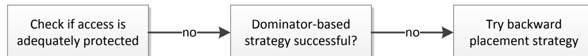


Figure 6: Analysis steps.

3.2 Valid Placement

Based on the challenges described in Section 2.2, we proceed to formulate what it means to have a *valid* placement of resource access prompts.

Definition 2 *We say that placement $P \subset N$ is a* valid placement *for a prompt placement problem $\mathcal{P} = \langle N, A, B, E, C, \mathcal{L} \rangle$ if the following conditions hold for every runtime execution of the app:*

- **Safe***: Every access to resource $r \in R$ is preceded by a prompt check for r.*
- **Visible***: No prompt is located within a background task or a library.*
- **Frugal***: Prompt for $r \in R$ is never invoked unless it is either followed by a call to* `get`(r) *or an exception occurs*[2]*.*
- **Not-repetitive***: Prompt for permission $r_2 \in R$ is never invoked if permissions for $r_1 \in R$ have already been granted and $r_2 \sqsubseteq r_1$ (that is, r_1 is at least as or* more *permissive than r_2).*

3.3 Solution Outline

We provide intuition for our solution in the remaining sections; Section 4 gives the actual algorithms. Figure 6 shows the overall flow of our analysis. Given a graph with well-identified resource access points, a *safe* placement is relatively easy to come up with. The main obstacle is the fact that we cannot always put prompts right before accesses, because sometimes accesses are within background tasks or, more frequently, in libraries (violating the *visible* requirement).

Intuitively, we can start with resource access points A and move the prompts up until we are outside of background tasks. The downside of this approach is a possibility of moving these prompts too far (to the beginning of the app in the most extreme case), which would violate the *frugal* requirement. This gives rise to a notion of a prompt being *needed* at a particular point, for which we use the term *anticipating*, common in compiler literature [4]. By way of example, for the code snippet in Figure 7, location access is anticipating before line 3,

[2]Note that this notion of frugality is optimized for runtime savings, not necessarily savings in terms of code size.

but it is *not* anticipating before the `if` on line 2, because of the `else` branch. So placing the prompt on line 1 leads to unnecessary prompting, violating the requirement of being *frugal*.

```
1.
2.  if(P){
3.      var l = getLocation();
4.  } else {
5.      x++;
6.  }
```

Figure 7: Conditional location access.

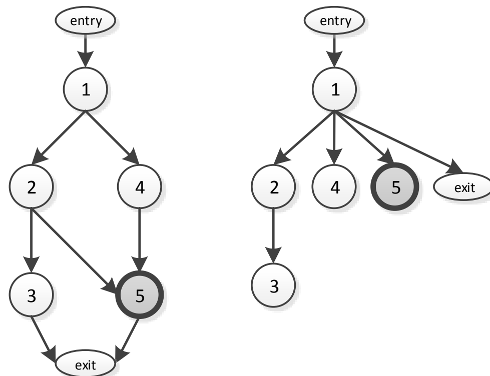


Figure 8: Graph (left) and its dominator tree (right). Node 5 is a resource access node within a library.

Definition 3 *We say that basic block $B \in N$ is r-*anticipating *if every path from B to $N_{\mathbf{exit}}$ passes through a resource access of type r.*

Intuitively, placing prompts for resource accesses of type r at r-anticipating nodes is necessary because these nodes are guaranteed to require them eventually; in other words, these placements will be *frugal*.

Finally, the discussion so far has not considered the case of prompts granting permissions of different "strength", resulting in potentially unnecessary prompts. This suggests that the notion of being anticipating should be defined not globally, but with respect to a particular kind of resource, taking into account the lattice of resource access permissions.

Dominator-based Approach: Using the notion of *dominators* in the graph [4] we can abstract away unnecessary details. Recall that we say that node $d \in N$ *dominates* node $n \in N$ if every path from $N_{\mathbf{entry}} \to n$ passes through d. Dominator relationships induce a dominator tree over the set of nodes N. An example of such a dominator tree for a graph in Figure 8a is shown in Figure 8b.

By this definition, dominator-based placement is an easy way to "block" access to a particular resource access. The most immediate approach is to place prompts on the nodes dominating the resource access node.

Of course, since we want a placement as close as possible to the access point, we will prefer the *immediate* dominator of the resource access node. By definition, we will have a *safe* placement, because $\forall\ a \in A$, every path from $N_{\mathbf{entry}} \to a$ must pass through $idom(a)$, the immediate dominator of a. This simple approach suffers from two problems:

- Background and library nodes can invalidate immediate dominator-based placement. To deal with the issue of visibility, we can shift the prompts up in the dominator tree.
- Immediate dominator-based placement can violate the *frugal* condition. Indeed, consider the location access at line 3 in Figure 7. Its immediate dominator is the `if(P)` node. However, this node is not location-anticipating, because the `else` branch is *not* accessing the location.

A viable approach is therefore to start at the resource access node and walk up the dominator tree until we encounter a node that is not in the background or a library. We are guaranteed to encounter such a node eventually, because sooner or later we will encounter $N_{\mathbf{entry}}$, which is a foreground non-library node by Definition 1.

For the graph in Figure 8, node 5 is a library node. Nodes 1 and $N_{\mathbf{entry}}$ are in the cover for node 5. Node 1 is the immediate cover of 5. Our approach, therefore, will choose node 1 for a prompt protecting node 5, but, unfortunately, this placement will violate the frugality condition.

Backward Placement: Sometimes dominator-based placement will backtrack "too far" in the graph to become unnecessary — in other words, not *frugal*. In these cases, we propose an alternative strategy called *backward placement*, which often avoids this problem. Backward placement explores the predecessors of the resource access node and find an individual *separate* place for a prompt for each of them. For node 5 in Figure 8, both predecessors 2 and 4 present valid placement opportunities, which are also frugal. Frequently, the backward placement approach will yield a valid placement. The concern with this strategy is two-fold:

- This approach may not scale well, as it involves an exponential graph search. While this is true

in general, in practice we frequently find a valid placement within several nodes, as detailed in Section 5.

- More prompts will be created compared to the dominator-based approach. (Indeed, in our Figure 8 example, we inserted two nodes instead of one.) More inserted prompts may increase the size of the rewritten app and may also make manual validation of placement results more challenging and time-consuming.

Unlike dominator-based placement, there is a possibility of passing through prompt placement nodes multiple times at runtime. To see this, consider adding a backward edge from $3 \rightarrow 1$ in Figure 8. This edge does not affect the dominator tree or dominator-based placement. If we place prompts at nodes 2 and 4 for resource access at node 5, there is a possibility of encountering the prompt at node 2 multiple times as we go through the loop $1 \rightarrow 2 \rightarrow 3$. This kind of double-prompting violates the not-repetitive condition in Definition 2. A simple way to address this is to record user consent in app's isolated storage for both the current runtime session and future app invocations, as shown in Section 4.3.

3.4 Placement Algorithm

In our evaluation section, we will examine the trade-offs between the dominator-based and backward placement strategies. To summarize, this is an outline of our placement approach:

1. For every $r \in R$ and every node $n \in N$, compute its r-anticipating value $A_r(n)$.
2. Merge values by meeting them in the semi-lattice $\mathcal{L} = \langle \mathcal{R}, \wedge \rangle$ for all resource types:

$$A(n) = \bigwedge_{r \in R} A_r(n)$$

3. For every resource access a of type r, use a backward search to find if it is adequately covered by existing prompts of type r' such that $r \sqsubseteq r'$.
4. If not, proceed to insert a prompt of type $A(n)$ using either a dominator-based or a backward placement strategy.

Anticipating values can be calculated using a simple data-flow computation, in the style of the Dragon book [4]. A formulation of this analysis is shown in Figure 10 in the form of a table traditional for succinctly representing data-flow problems. The advantage of such a formulation is that it runs in linear time, given a lattice of finite height (and size),

```
1: function InsertPrompt(G, a, ant, idom)
2:   if ¬HasPrompt(G, a.Type, a) then
3:     // Try dominator-based first
4:     Placement ← ∅
5:     success ← InsertPrompt-D(G, a, ant, idom)
6:     if ¬ success then
7:       // Try backward placement next
8:       Placement ← ∅
9:       InsertPrompt-B(G, a, ant)
10:    end if
11:  end if
12: end function
13:
14: //Dominator-based placement
15: function InsertPrompt-D(G, a, ant, idom)
16:   n ← a
17:   while n ≠ N_entry do
18:     if IsAnticipating(n, a.Type, ant)  ∧
19:         n ∉ G.Background  ∧  n ∉ G.Libraries
20:     then
21:       Placement ← Placement ∪ {n}
22:       return true
23:     else
24:       n ← idom(n)   ▷ Proceed to the immediate dominator
25:     end if
26:   end while
27: return false
28: end function
29:
30: //Backward search placement
31: function InsertPrompt-B(G, a, ant)
32:   Occurs-check(a)               ▷ Prevent infinite recursion
33:   if ¬IsReachable(a)  ∨  (IsAnticipating(a, a.Type, ant)
34:         ∧  a ∉ G.Background  ∧  a ∉ G.Libraries)
35:    then
36:     Placement ← Placement ∪ {a}
37:     return true
38:   else
39:     for all p ∈ G.predecessors(a) do        ▷ Predecessors
40:       success ← InsertPrompt-B(G, p, ant)
41:       if ¬success then
42:         return false          ▷ One of the predecessors failed
43:       end if
44:     end for
45:     return true                     ▷ All predecessors succeeded
46:   end if
47: end function
48:
49: //Helper function to check if n is anticipating for r ∈ R
50: function IsAnticipating(n, r, ant)
51:   r' ← ant(n)                 ▷ Computed prompt type at n
52:   return r ⊑ r'               ▷ True if r' is more permissive
53: end function
```

Figure 9: Insertion of resource access prompts. G is the graph; a is the access node; $ant : N \rightarrow 2^R$ is the anticipating lookup map computed as specified in Figure 10, and, finally, $idom$ is the immediate dominator relation.

Semi-lattice	L	2^R, the power set of R	
Top	$\top$	$\emptyset$	
Initial value	$init(n)$	$\emptyset$	
Transfer func.	$TF(n)$	add r to set	if n is an access for $r \in R$
		identity	otherwise
Meet operator	$\wedge(x, y)$	union $x \cup y$	
Direction		*backward*	

Figure 10: Dataflow analysis formulation for computing anticipating nodes: $\forall n \in N$, we compute the set of resource types that node n is anticipating.

```
1: //Checks for existing prompts
2: function HasPrompt(G, r, a)
3:   Occurs-check(a)                          ▷ Prevent infinite recursion
4:   if a ∈ G.Prompts then
5:     r′ ← a.Type
6:     adequate ← (r ⊑ r′)       ▷ Existing prompt at least as
   permissive as needed?
7:     if adequate then
8:       return true               ▷ Check if adequately protected
9:     end if
10:  end if
11:
12:  //Explore all predecessors in turn
13:  for all p ∈ G.predecessors(a) do
14:    success ← HasPrompt(G, r, p)
15:    if ¬success then
16:      return false              ▷ One of the predecessors failed
17:    end if
18:  end for
19:  return true                       ▷ All predecessors succeeded
20: end function
```

Figure 11: Checking for resource access prompts. G is the graph; r is the resource type; a is the access node.

and that most compiler frameworks already provide a data-flow framework into which this kind of analysis can be "dropped".

There is some flexibility when it comes to the last step. Indeed, we can choose to use a dominator-based, or a backward placement strategy, or some combination. In our implementation, we try the dominator strategy first to see if it yields a valid placement and, failing that, resort to the backward strategy. This hybrid approach is shown in the function InsertPrompt in Figure 9. Note that if placement is successful, the outcome is stored in the $Placement \subset N$ set.

InsertPrompt-B has an occurs-check on line 32 to avoid the possibility of infinite recursion for graphs with loops, which are encountered in the process of backward exploration. If the current node is not reachable from non-library code as indicated by *IsReachable*, we return *true*. We discuss the challenges of fast backward computation in Section 4.2.

3.5 Checking For Existing Prompts

Note that before we choose to insert prompts we need to make sure they are in fact missing as shown on line 2 of Figure 9. Doing so requires a backward search, as shown in Figure 11. Note that in practice, HasPrompts frequently returns *false*, failing quickly without exploring the entire set of predecessors. Section 4.2 demonstrates how this search can be made faster.

3.6 Proof Sketch

The algorithm that pulls everything together to create a placement is shown in Figure 12. We first check that whether there is indeed a valid placement for all resource accesses. Once this is ensured, we proceed to modify the underlying graph by inserting prompts at appropriate places. Note that prompt insertion is only attempted if they are in fact missing, as ensured by the check on line 2 of Figure 9. The details of runtime instrumentation are given in Section 4.3. The structure of the algorithm allows us to reason about the resulting placement.

Theorem 1 *The placement of prompts above is in fact valid if the placement routine* CreatePlacement *returns true.*

Proof sketch: It is easier to consider each correctness property in turn. We will refer to code lines in Figure 9 unless indicated otherwise.

Safe: We need to ensure that every access a to resource r is preceded by a prompt check for r. The call to InsertPrompt must have returned true for resource access a. This is because either the dominator-based or backward strategy was successful. If the dominator-based strategy succeeded, there was a non-background, non-library node dominating a which is also anticipating for a.Type. The check on line 18 maintains this invariant. If the dominator-based strategy failed and the backward strategy succeeded, this is because *every* path from a to $N_{\mathbf{entry}}$ has encountered a placement point which satisfied the check on line 33, providing adequate protection for the access at a.

Visible: No prompt is placed within a background task or library code. This is true by construction because of checks on lines 19 and 34.

Frugal: Placement only occurs at anticipating nodes because of checks on lines 18 and 33.

```
1: function CreatePlacement(G, ant, idom)
2:   for all a ∈ G.Accesses do
3:     success ← InsertPrompt(G, a, ant, idom)
4:     if ¬ success then
5:       return false
6:     else
7:       for all p ∈ Placement do
8:         Prompts ← Prompts ∪ ⟨p, ant(a)⟩
9:       end for
10:    end if
11:  end for
12:
13:  // All clear: proceed with the placement
14:  for all ⟨n, t⟩ ∈ Prompts do
15:    InsertAtNode(n, t)
16:  end for
17:  return true
18: end function
```

Figure 12: Putting it all together: creating an overall prompt placement for graph G.

Not-repetitive: Prompt for $r_2 \in R$ is never invoked if permissions for r_1 have already been granted and $r_2 \sqsubseteq r_1$. This property is maintained by a combination of three steps: (1) merging in Step 2 on the overall algorithm, (2) check on line 52 *and* (3) the runtime "sticky" treatment of prompts that avoids double-prompting for the same resource type further explained in Section 4.3.

4 Implementation Details

Our current implementation of the static analysis described in this paper involves dealing with a variety of practical details, some of which are fairly common in bytecode-based static analysis tools, whereas others are quite specific to our setting of WP apps written in .NET.

A significant part of the implementation involves building a graph on which to perform our analysis. Intra-procedurally, we parse the .NET bytecode to construct basic blocks; we terminate them at method calls to simplify analysis. For call graph construction, we use a simple class hierarchy analysis (CHA) to resolve virtual calls within the program. We also construct a dominator tree as part of graph construction, as we need it later. In many cases, the resulting graphs have enough precision for our analysis.

4.1 Reflection & Analysis Challenges

WP applications are distributed as XAP files, which are archives consisting of code in the form of bytecode DLLs, resources such as images and XAML, and the app manifest, which specifies requested capabilities, etc. Unsurprisingly, various reflective constructs found in WP apps create challenges for our analysis. While we outline some of the details of our solutions below, constructing precise static call graphs for mobile apps remains an ongoing challenge, and require further research.

Analysis imprecision usually does not stem from the underlying call graph construction approach, which could be alleviated through pointer analysis, which generally provides sufficient precision for call graph construction, but in challenges specific to complex WP apps, as discussed below.

Event handlers: The code below illustrates some complications posed by event handlers.

```
static void Main(string[] args) {
  AppDomain.CurrentDomain.ProcessExit +=
            new EventHandler(OnProcessExit);
}

// library code
static void OnProcessExit(object sender, EventArgs e) {
  // location access
  var watcher =
    new System.Device.Location.GeoCoordinateWatcher();
  var pos = watcher.Position;
}
```

By default, method `OnProcessExit` does not have any predecessors in the call graph. At runtime, it may in fact be called from a variety of places, which is not easy to model as part of call graph construction. However, it may not be called *before* the event handler is registered in method `Main`. Our solution is to augment the call graph construction code to create a special invocation edge from the registration site to `OnProcessExit`. The analysis will then be able to place the prompt right before the registration in method `Main`, which makes a significant difference in our ability to find successful placements.

Actions and asynchronous wrappers: Another similar form of delayed execution in WP apps is *actions* (`System.Action`) and its asynchronous cousin `System.AsyncCallback`, which are effectively wrappers around delegates registered for later execution. We deal with actions in a way that is similar to event handlers.

XAML: A particular difficulty for analysis stems from the use of declarative UIs specified in XAML, an XML-like language that combines an easy-to-read UI specification with "hooks" into code. XAML is compiled into special resources that are embedded into an app's DLLs. When the method `InitializeComponent()` is called on the class specified in XAML, it proceeds to register events that are specified declaratively, as shown in a XAML snippet below:

```
1  <phone:PhoneApplicationPage.ApplicationBar>
2     <shell:ApplicationBar IsVisible="True">
3       <shell:ApplicationBar.MenuItems>
4         <shell:ApplicationBarMenuItem Text="Settings"
5             Click="SettingsClick" />
6       </shell:ApplicationBar.MenuItems>
7     </shell:ApplicationBar>
8   </phone:PhoneApplicationPage.ApplicationBar>
```

Event handler `SettingsClick` should be properly registered so that it can later be invoked.

Alas, some aspects of declarative app specification defy static analysis. A typical example is navigation between an app's pages.

```
1  base.NavigationService.
2      Navigate(new Uri(
3              "/VenueByGeo.xaml?mc=" + this.strMenuCode +
4              "&t=" + this.strToken,
5              UriKind.RelativeOrAbsolute));
```

Statically, we do not know which page will be navigated to, and, consequently, which `OnNavigatedTo` event handler will be called. To avoid polluting the call graph, we only link up page navigation when

the destination is a string constant. Unfortunately, this approach is unsound. A more robust technique would be to integrate a string analysis [8, 19, 33] into our implementation.

Summary: Reflective coding constructs are the Achilles heel of static analysis. While this is true as it applies to applications written in .NET and Java, this is especially so given the declarative programming style often used in WP apps, where code is "glued together" with declarative specification. Several approaches to handling reflection have been proposed and used in the literature [6, 18, 26, 28, 35]. Alas, all of them require a certain degree of customization to the problem and APIs at hand. Additionally, reflection analysis tends to be intertwined with a heavyweight analysis such as a points-to. We instead opt for a lightweight analysis that pattern-matches for the easily-to-resolve case, potentially introducing unsoundness. We evaluate the effects of this treatment in Section 6.

```
[SomaAd..ctor() @ 0134) bg      // resource access
 [SomaAd..ctor() @ 0120) bg
  [SomaAd..ctor() @ 0118) bg
   [SomaAd..ctor() @ 0000) bg
    [SomaAdViewer.StartAds() @ 00a6) bg
     [SomaAdViewer.StartAds() @ 009e) bg
      [SomaAdViewer.StartAds() @ 0000) bg
       [CollectHome.g_AdFailed(object, ...) @ 00f7) fg
        [CollectHome.g_AdFailed(object, ...) @ 0052) fg
         [CollectHome.g_AdFailed(object, ...) @ 000a) fg
          [CollectHome.g_AdFailed(object, ...) @ 0000) fg
          [CollectHome.g_AdFailed(object, ...) @ 0040) fg
           [CollectHome.g_AdFailed(object, ...) @ 0030) fg
            [CollectHome.g_AdFailed(object, ...) @ 0008) fg
          [CollectHome.g_AdFailed(object, ...) @ 004a) fg
           [CollectHome.g_AdFailed(object, ...) @ 00df) fg
          [CollectHome.g_AdFailed(object, ...) @ 006c) fg
           [CollectHome.g_AdFailed(object, ...) @ 0066) fg
```

Figure 13: A backward exploration tree of depth 20. Method names and signatures are abbreviated for brevity. `bg` and `fg` stands for background/library vs. foreground/non-library methods, respectively.

4.2 Fast Backward Placement

Recall from Section 3 that our approach resorts to a search for both checking if a resource access is already protected with a prompt and for inserting prompts if the dominator-based strategy fails. In implementing backward search, we need to be concerned with preventing infinite recursion (the occurs-check from Section 3). There is also the possibility of exponential path explosion, which is quite real given that we we are dealing with graphs that typically have tens of thousands of nodes. It is therefore imperative to design an efficient exploration strategy.

Our approach for both checking for prompts and inserting them relies on first building a *spanning tree* rooted at the access node, computed using a depth-first search. Figure 13 gives an example of such a tree. The tree allows us to classify underlying graph edges as ether forward, backward, or cross edges. Further analysis is performed on the tree as a series of downward passes, implemented as recursive procedures, starting at the resource access and exploring the predecessors[3]. In summary, we perform three recursive passes over the spanning tree. Each pass computes a boolean value for each of the visited nodes to represent the checking or placement status; values are maintained across the passes in a map called v.

The advantage of this multi-pass approach is its simplicity and guaranteed runtime complexity. We start with all spanning tree nodes as unvisited and then perform three recursive traversals of the tree, as shown in Figure 14 and described below. In our implementation, we reuse the same spanning tree for the prompt checking and placement analysis stages. This approach is linear in the size of the graph, and is generally quite fast, even when there are hundreds of nodes reachable from a resource access.

1. **Traverse:** For each non-library non-background node, declare it as a valid placement point and set $v[n]$ to *true*[4]. For other nodes, if *all* their children have their v as *true*, set $v[n]$ to *true*; otherwise, set $v[n]$ to *false*.
2. **Patch-up:** Traverse the tree considering cross-edges originating at the current node. If all cross-edges emanating from nodes have valid placements (v value is *true*), set $v[n]$ to *true*.
3. **Collect:** Propagate (newly) *true* values up to the root: set $v[n]$ to *true* if the v value is *true* for all of n's children.

The final result is computed by running all three steps in order and examining the result at the root of the spanning tree.

4.3 Runtime Considerations

While much of the focus of this paper is on statically locating placement points, choosing the right kind of runtime instrumentation presents some interesting

[3] To avoid stack overflow issues stemming from deep trees, once the tree has been constructed, we make sure that the size is below a fixed threshold (set to 250 for our experiments).

[4] Note that to maximize backward placement opportunities, for all unreachable nodes, we set $v[n]$ to *true*, as shown in Figure 9. This is because the presence of dead code should not prevent prompt placement.

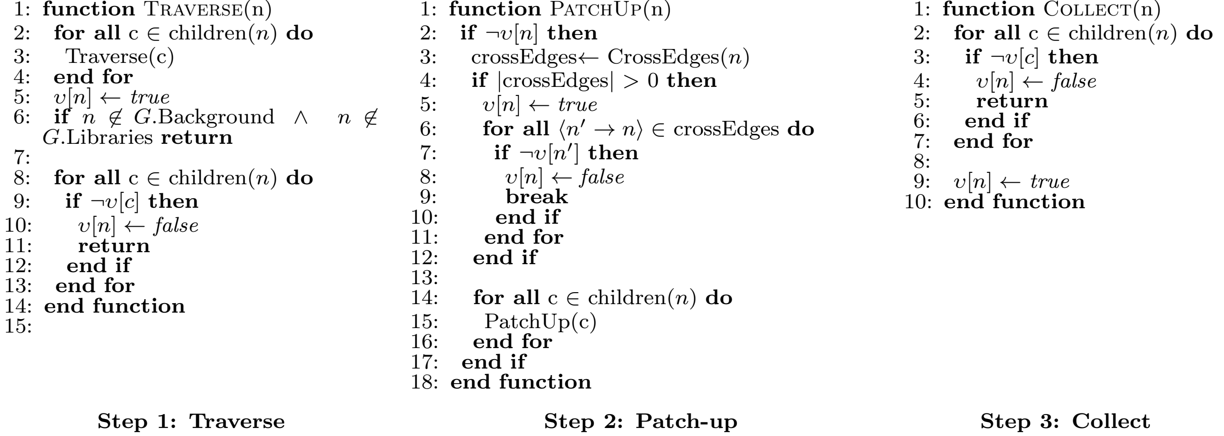


Figure 14: Three-stage backward placement algorithm explained in Section 4.2.

challenges. We need to ensure that we are not going to induce double-prompting, as discussed in Section 3. To do so, we maintain a "sticky" app-global setting value in the app's isolated storage, as illustrated by the following example for the fine-grained GPS location resource type:

```
1  var setting = IsolatedStorageSettings.
2       get_ApplicationSettings().
3           get_Item("UserLocationSettings");
4  if (setting == null){
5   int result = MessageBox.Show(
6     "Is it okay to access your fine-grained GPS location?",
7     "Allow "+Assembly.GetExecutingAssembly().FullName()+
8                     " to access and use your location.",
9     1);
10   {
11     settings.set_Item("UserLocationSettings",
12                           (result == 1) ? "Y" : "N");
13   }
14  }else{
15    if(setting.ToString().Equals("Y")){
16        // proceed with the prompt
17    }
18  }
```

Because the prompt remains sticky application-wide and persists across application invocations, even if we conservatively *insert* an extra prompt, we will only *show* it at most once per app.

5 Evaluation

We have analyzed 100 WP 7 apps from the WP Store to collect our results. To make the analysis more meaningful, we have selected only apps with `LOCATION` and `NETWORKING` capabilities. Such apps constitute about a fifth of a larger set of about 2,000, from which we drew our 100 app sample. The goal of our evaluation is to understand how frequently prompts are omitted and to attempt to insert prompts in a fully automatic manner.

Characterizing the input: We first present some aggregate statistics of the analysis results in Figure 15. WP applications are quite substantial in size, constituting about 3,528 methods on average. This is in part because they rely (and therefore recursively include within their call graph) large libraries, some of which are part of the operating system SDK, and others are included .NET libraries. The average size for our apps is 7.3 MB; many consist of dozens of DLLs.

We discovered that the libraries shown in the inlined figure are included most frequently. These libraries provide advertising functionality, and many request location data. About 7% of all methods are contained in background tasks or libraries, which presents a significant challenge for prompt placement. Out of these, most are in fact in third-party libraries. Recall that we do not want to place prompts in libraries. To recognize third-party libraries in our experiments, we used a list of 100 common advertising libraries, identified by the DLL in which they are contained; these include `Microsoft.Advertising.Mobile.dll`, `AdRotator.dll`, `MobFox.Ads.LocationAware.dll`, `FlurryWP7SDK.dll`, `Inneractive.Nokia.Ad.dll`, `MoAds.dll`, `adMob7.dll`, `Photobucket.Ads.dll` and many others. Our analysis is parameterized with respect to this list. Frameworks such as these may access GPS location deep within library code, making prompt placement

Component	Count
SOMAWP7	42
NetDragon.PandaReader	13
EchoEchoBackgroundAgent	10
Utilities	10
BMSApp	10
MobFox.Ads.LocationAware	8
XIMAD_Ad_Client	7
EchoEcho	5
DirectRemote	5
DCMetroApp	5

apps analyzed	100
processed methods	352,816
background/library methods	26,033
library methods	25,898
nodes	1,333,056
anticipating	171,253
accesses	227
accesses in background/library methods	78

Figure 15: Apps analyzed: summary of input statistics.

succeeded	202
failed	19
succeeded unique	143
failed unique	7
dominator-based succeeded	150
naïve	143
backward succeeded	56
regular	150
dead code	2,094
backward placements	(40,270, 56)
depth exceeded	15

Figure 16: Prompt placement: summary of results of applying analysis to 100 apps.

analysis particularly difficult.

Our analysis represents each application as 13,330 nodes on average. Out of these, about 12% are considered to be anticipating by our analysis. In other words, about 88% of nodes are *not* eligible prompt placement points.

The last section of Figure 15 describes the resource accesses found in these 100 applications. Across all apps, there are 227 resource accesses we analyze. Overall, apps have an average of 2.27 resource accesses, with a maximum of 9 for one of the apps. The figure shown inline in this paragraph shows how frequent individual resource types are. We find that the majority of sensitive resource accesses are to GPS location data, with occasional accesses to user contacts and calendar.

Location	95.15%
Contacts	4.41%
Calendar	0.44%

Inserting prompts: Figure 16 provides statistics describing the prompt placement process. Overall, our two-prong strategy of dominator-based and backward placement succeeds in about 91% of all cases. However, it is important to observe that many cases, including challenging resource accesses deep in library code, are shared by many applications. To avoid double-counting, we show the number of *unique* placement attempts that have succeeded and failed. Considering these numbers of unique accesses, we are able to successfully place prompts in 95% of cases (143 out of 150), a higher success percentage. Several other lessons can be drawn from the rest of the table:

- When dominator-based placement succeeds, it is usually immediate (95% of all dominator-based successes are *naïve* successes).
- Backward placement is helpful for cases where dominator-based placement fails. However, some of these cases are still too hard, leading to 7 unique failures.

	Average	**Max**	**#**
app loading	1,779	24,585	100
call graph construction	18,152	147,287	100
placement graph construction	15,103	293,480	100
anticipating computation	158	3826	86
finding missing prompts	123	649	100
prompt insertion, per app	942	70,228	103
dominator-based, per access	0.05	1	221
backward, per access	1,366	49,277	71

Figure 17: Timing, in *ms*. All measurements are per app, unless stated otherwise.

Timing: Figure 17 provides a summary of timing information for our analysis. For each measurement, we provide the average timing across 100 apps, the maximum observed time and the number of observations. Each measurement in given in *ms*. Overall, the most time goes into initial processing of the application, which involves reading it from disk, constructing a representation of the app's assemblies in memory (1.7 seconds on average), traversing it to create a call graph and control flow graphs (CFGs) (18 seconds on average), dominator calculation, and reachability calculation, resulting in a graph suitable for analysis. Computing anticipating nodes only takes 158 *ms* on average.

Finding missing prompts takes about 123 *ms* on average, in part because many instructions need to be examined in search of existing prompts. Prompt insertion, on average, is fast, only about .9 seconds per application. Dominance-based placement is virtually instantaneous. Backward placement is slower, at 1.3 seconds per resource access, raising the average. Based on these performance numbers, we are optimistic that prompt insertion can be done entirely automatically over a large number of applications.

6 Discussion

We have selected static analysis as a method of choice to avoid code coverage issues inherent with runtime analysis and for analysis speed (end-to-end processing is several minutes per app). In this section we discuss some of the limitations of our current static analysis approach. There are two potential sources of errors in our analysis. Our analysis may classify a resource access as *unprotected* whereas it is properly protected with runtime prompts; we call these cases *false positives*. By the same token, our analysis may classify a resource access as *protected* whereas in fact at runtime there are no preceding prompts that protect the resource access; we call these cases *false negatives*.

Manual inspection: We examined a subset of applications to manually check for these errors. The verification process includes running these applications in the emulator to collect network packets and to collect API calls invoked by each application at runtime. We manually exercise as much functionality of each application as possible. If the application presented a runtime prompt, we inspected the text of the message and clicked through each "allow" (to use my location) and "don't allow" button to determine how the choice affects application behavior.

Once the runtime inspection was complete, we examined network packets and invoked API lists, correlating them with the app's disassembled code to verify the observed behavior. Although this verification process is thorough, it requires significant manual efforts, thus limiting the number of cases that can be examined. Next, we discuss findings from 10 applications. These apps contain 27 resource access points, among which 21 are classified as *unprotected* by our analysis.

6.1 False Negatives

Our manual analysis found no false negatives. On a close examination of each of the 27 resource accesses, we find 10 accesses that are not protected. Our analysis correctly identifies all of these accesses as unprotected and finds proper placements.

These unprotected accesses are found in third-party libraries included across 5 apps. Interestingly, in an effort to maximize revenue, one app embeds *two* advertising related third-party libraries (`SOMAWP7.dll` and `AdRotatorXNA.dll`) and *both* contain unprotected location accesses. Two placements are made via dominator-based placement; the other eight through backward placement. Backward placements result in 40 inserted prompts in application code, which upon casual examination appear to be correct. We find these results promising, as users express increasing concern about data sharing with third parties [21], and our analysis properly detects and fixes such unprotected accesses.

6.2 False Positives

Eleven out of 21 accesses flagged as unprotected turn out to be properly protected. Although the number of false positives is somewhat high, with manual inspection, we found the following reasons for them:

Sticky location prompt: Seven false positives are due to our analysis's inability to analyze sticky location prompts, as shown in Figure 5. Three cases are similar to the example in Figure 18(a). The rest are caused by one application that uses the location flag to enable or disable the button that allows the user to navigate to the page (that invokes location access) as shown in Figure 18(b). WP apps can use several different storage mechanisms; we are looking into ways to detect them statically.

Consent dialog implementation: Two false positives are due to the limitation of identifying existing prompts. Both result from a single app that implements a custom consent dialog page instead of `MessageBox()`, as shown in Figure 19. We are looking into ways to parse a blocking page with buttons to detect such custom-made consent dialog pages, although this is obviously a difficult problem. However, such cases are not common and we find that five out of six applications that show prompts employ `MessageBox()`, as expected.

Async calls and `XAML` files: Two false positives are due to limitations of call graph construction. Figure 20(a) shows an expanded example of the case discussed in Section 4.1. Applications may use multiple types of `EventHandler`s to be called asynchronously. In our current implementation, we parse `EventHandler`s and add links when handlers are registered. However, the current implementation fails when multiple delegates and `EventHandler`s are used in a tricky way, as shown in Figure 20(b). We are investigating ways to extend our call graph construction to support these cases.

6.3 Effect of False Positives

Like most practical static analysis tools, our analysis is potentially vulnerable to false positives, primarily because of program representation challenges. Unlike most static analysis tools for bug detection, our analysis is two-phase: if it detects that a resource

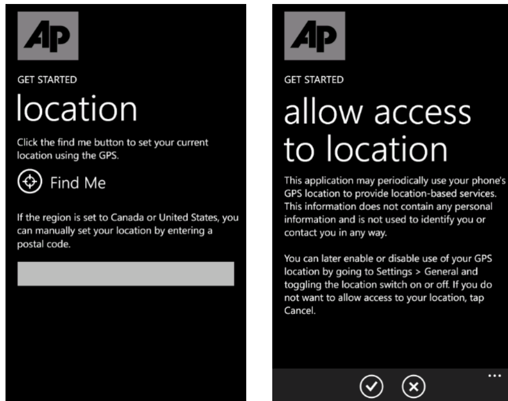
```
private void mapLocaitons() {
  if (this.avisAppUnitService.UseLocationsMapping) {
    this.watcher=null;
    GeoCoordinateWatcher watcher=new GeoCoordinateWatcher
     (GeoPositionAccuracy.Default){MovementThreshold = 20.0};
    this.watcher = watcher;
    ...
    this.watcher.Start();
  }
}
```

(a) Sticky prompt example #1: This app saves the result of the prompt response in `Athis.avisAppUnitService.UseLocationsMapping`.

```
public MapPage() {
  this.InitializeComponent();
  base.DataContext = new MapViewModel();
  this.BuildApplicationBar();
  if (AppSettings.Current.UseLocationService) {
    this.watcher = new GeoCoordinateWatcher();
  }
  ((ApplicationBarIconButton)base.ApplicationBar.Buttons[0]).
   IsEnabled = AppSettings.Current.UseLocationService;
  ((ApplicationBarIconButton)base.ApplicationBar.Buttons[2]).
   IsEnabled = AppSettings.Current.UseLocationService;
   this.UpdatePushpinsBackground();
}
```

(b) Sticky prompt example #2: This app disables page navigation based on the location access depending on `AppSettings.Current.UseLocationService`.

Figure 18: Sticky prompt examples.



(a) App page with location access. (b) Prompt (consent dialog).

Figure 19: False positive due to a custom prompt: A prompt is customized as a separate WP UI page.

access is not adequately protected, it tries to propose a placement of prompts that would protect it. Our analysis errs on the safe side, introducing false positives and not false negatives.

False positives, however, may lead to double-prompting, since our analysis will inject a prompt to protect already protected resource accesses. Because our inserted prompts are sticky, our approach introduces at most one extra runtime prompt per app during the entire app's lifecycle, which we be-

```
private void GPS_MouseLeftButtonDown(object sender,
    MouseButtonEventArgs e) {
  ...
  else if (MessageBox.Show("Sharing this info allows us to
    find theaters and events near you.We won't share
    this information.", "Allow BookMyShow to access and
    use your location.",MessageBoxButton.OKCancel)==
    MessageBoxResult.OK) {
    ...
    base.NavigationService.Navigate(
        new Uri("/VenueByGeo.xaml?mc="
    + this.strMenuCode + "&t=" + this.strToken,
    UriKind.RelativeOrAbsolute));
  }
  ...
}
```

(a) Complex CFG #1: Function `Navigate()` internally calls `BMSApp.VenueByGeo.OnNavigatedTo()` as defined in `VenueByGeo.xaml`.

```
public static bool GetCurrentLocation() {
  ...
  Observable.FromEvent
   <GeoPositionStatusChangedEventArgs>
   (delegate
     (EventHandler<GeoPositionStatusChangedEventArgs> ev){
       GeoCoordinateWatcher.StatusChanged += ev;
      }, delegate (EventHandler
        <GeoPositionStatusChangedEventArgs> ev) {
      GeoCoordinateWatcher.StatusChanged -= ev;
      }).Where<IEvent
       <GeoPositionStatusChangedEventArgs>>
       (delegate ... args){
      if (args.EventArgs.Status != GeoPositionStatus.Ready)
      {
        return (args.EventArgs.Status ==
          GeoPositionStatus.Disabled);
       }
      return true;
    }).Take<IEvent<GeoPositionStatusChangedEventArgs>>(1).
      Subscribe<IEvent<GeoPositionStatusChangedEventArgs>>
      (delegate
      (IEvent<GeoPositionStatusChangedEventArgs> args) {
      if (args.EventArgs.Status == GeoPositionStatus.Ready)
      {
        RaiseCurrentLocationAvailable(
          new CurrentLocationAvailableEventArgs(
            GeoCoordinateWatcher.Position.Location));
      }
 ...
}
```

(b) Complex CFG #2: This code generates a compiler-generated function `<GetCurrentLocation>b__3` in `Eventful.Helpers.LocationHelper`, which is called within `GetCurrentLocation()`, as defined in `VenueByGeo.xaml`.

Figure 20: Complex CFG cases.

lieve will not lead to prompt fatigue. Nonetheless, double-prompting can trigger confusion in end-users and therefore should be minimized. Our experience with the ten test applications shows that in all cases, resource accesses get triggered quickly, with several clicks, so runtime checking of this kind is unlikely to require excessive effort. If desired, runtime testing by the developer or App Store maintainers can accompany our analysis to detect and eliminate potential double-prompting.

7 Related Work

The requirement of protecting privacy-sensitive resource accesses with runtime prompts or consent dialogs has only recently been introduced to mobile applications. To our knowledge, no previous work has investigated static analysis approaches to detect unprotected resource accesses in mobile application binaries. This section discusses previous research in three related areas: automatic hook placement, graph-based analysis for information security, and user studies of consent dialogs.

Automatic hook placement: A number of previous studies examine the issues of protecting security-sensitive operations with authorization hooks (e.g., checking permissions for file operations). Ganapathy *et al.* [14] use a static program analysis over the Linux kernel source code to identify previously unspecified sensitive operations and find the right set of hooks that need to protect them. AutoISES by Tan *et al.* [34] is designed for the similar goal as [14] but the ways that AutoISES infers access to sensitive data structure are different from [14]. Muthukumaran *et al.* [27] focus on server code such as the X server and postgresql and use their insight concerning object access patterns in order to identify sensitive operations that require authorization.

In comparison to these efforts, our work begins with a set of known APIs that access sensitive resources. Such a set is easy to mine from developer documentation for most mobile operating systems. In particular, our work focuses on algorithms to find placements that meet the four important conditions specific to user prompts on mobile devices, whereas the previous work concentrates of placement being safe [14, 34] or safe and not-repetitive [27].

Graph-based analysis: Program dependence graphs are used for analyzing information security of programs in several projects [16, 17, 32]. Program dependence graphs include both data dependencies and control dependencies whereas the dataflow graphs that we use in this work typically contain just data dependencies. Hammer *et al.* [15] consider the enforcement of declassification [30] using program dependence graphs. Recent efforts focus on automating security-critical decisions for application developers [31, 36]. The use of a security type system for enforcing correctness is another case of co-operating with the developer to achieve better code quality and correctness guarantees [29]. Livshits and Chong [25] address the problem of sanitizer placement through static analysis and partially inspire our work on consent dialog placement. In our work, we use a backwards traversal to find the closest valid node to insert a missing prompt. Au *et al.* [5] use a similar backward reachability analysis over a call graph constructed from the Android framework. However, their goal is to create a mapping between API calls and permission checks and therefore their analysis need not consider the four conditions.

Mobile user privacy and consent dialogs: Several recent studies have investigated the effectiveness of existing consent dialogs used on mobile devices at informing users about which privacy-sensitive data can be accessed by apps. Felt *et al.* [12] show that only 17% of study participants paid attention to the permissions when installing Android applications. This finding may indicate that placing consent dialogs at install time (far removed from when the data is actually being accessed) renders these dialogs ineffective. On the contrary, a study by Fisher *et al.* focus on iPhone users' responses to *runtime* consent dialogs to location access and shows that 85% of study participants actually denied location requests for at least one app on their phone [13].

Although orthogonal to our work, previous studies have explored ways to improve the presentation of consent dialogs in mobile devices. Lin *et al.* measure users' "expectations" of apps' access to phone resources [23]. By highlighting unexpected behaviors in the Android permissions interface, the authors show that the new permission interface is more easily understood and efficient than the existing one. Felt *et al.* propose a framework for requesting permissions on smartphones [11]. Findings of these studies can inform a better usable privacy design of a consent dialog, which our analysis can automatically insert in mobile apps.

8 Conclusions

In this paper, we have explored the problem of missing prompts that should guard sensitive resource accesses. Our core contribution is a graph-theoretic algorithm for placing such prompts automatically. The approach balances the execution speed and few prompts inserted via dominator-based placement with a comprehensive nature of a more exhaustive backward analysis.

Overall, our two-prong strategy of dominator-based and backward placement succeeds in about 95% of all unique cases. Our approach is highly scalable; once the application has been represented in the form of a graph, analysis usually takes under a second on average.

References

[1] Pandora discloses privacy-related US inquiry into phone apps. http://www.nytimes.com/2011/04/05/technology/05pandora.html, April 2011.

[2] Daily report: Social app makes off with address books. http://bits.blogs.nytimes.com/2012/02/08/daily-report-social-app-makes-off-with-address-books/, February 2012.

[3] LinkedIn's iOS app collects and transmits names, emails and notes from your calendar, in plain text. http://thenextweb.com/insider/2012/06/06/linkedins-ios-app-collects-and-sends-names-emails-and-meeting-notes-from-your-calendar-back-in-plain-text/, June 2012.

[4] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.

[5] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *ACM CCS*, 2012.

[6] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 241–250, 2011.

[7] R. Böhme and S. Köpsell. Trained to accept?: a field experiment on consent dialogs. In *Proceedings of CHI*, 2010.

[8] A. S. Christensen, A. Møller, and M. Schwartzbach. Precise analysis of string expressions. In *International Conference on Static analysis*, 2003.

[9] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the Annual Network and Distributed System Security Symposium*, Feb. 2011.

[10] 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 Conference on Operating Systems Design and Implementation*, 2010.

[11] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner. How to ask for permission. In *Proceedings of HotSec*, 2012.

[12] A. P. Felt, E. Hay, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of SOUPS*, 2012.

[13] D. Fisher, L. Dorner, and D. Wagner. Short paper: location privacy: user behavior in the field. In *Proceedings of SPSM*, 2012.

[14] V. Ganapathy, T. Jaeger, and S. Jha. Automatic placement of authorization hooks in the linux security modules framework. In *ACM CCS*, 2005.

[15] C. Hammer, J. Krinke, and F. Nodes. Intransitive noninterference in dependence graphs. In *2nd International Symposium on Leveraging Application of Formal Methods, Verification and Validation*, Nov. 2006.

[16] C. Hammer, J. Krinke, and G. Snelting. Information flow control for java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering*, Mar. 2006.

[17] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, Dec. 2009.

[18] M. Hirzel, D. von Dincklage, A. Diwan, and M. Hind. Fast online pointer analysis. *ACM Trans. Program. Lang. Syst.*, 29(2), 2007.

[19] P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. In *Verification, Model Checking, and Abstract Interpretation*, pages 248–262. Springer, 2011.

[20] P. Internet. Privacy and data management on mobile devices. http://pewinternet.org/Reports/2012/Mobile-Privacy.aspx, September 2012.

[21] J. Jung, S. Han, and D. Wetherall. Short paper: Enhancing mobile application permissions with runtime feedback and constraints. In *Proceedings of SPSM*, 2012.

[22] S. Kaplan, B. Livshits, B. Zorn, C. Seifert, and C. Curtsinger. "nofus: Automatically detecting" + string.fromcharcode(32) + "obfuscated ".tolowercase() + "javascript code". Technical Report MSR-TR-2011-57, Microsoft Research, May 2011.

[23] J. Lin, S. Amini, J. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. Expectation and purpose: Understanding users' mental models of mobile app privacy through crowdsourcing. In *Proceedings of UbiComp 2012*, 2012.

[24] B. Livshits. Dynamic taint tracking in managed runtimes. Technical Report MSR-TR-2012-114, Microsoft Research, 2012.

[25] B. Livshits and S. Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *Proceedings of the Sympolisium on Principles of Programming Languages (POPL)*, Jan. 2013.

[26] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *Asian Symposium on Programming Languages and Systems*, Nov. 2005.

[27] D. Muthukumaran, T. Jaeger, and V. Ganapathy. Leveraging "choice" to automate authorization hook placement. In *ACM CCS*, 2012.

[28] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *ACM Sigplan Notices*, volume 45, pages 1–12. ACM, 2010.

[29] W. Robertson and G. Vigna. Static enforcement of Web application integrity through strong typing. In *Proceedings of the Usenix Security Symposium*, Aug. 2009.

[30] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 255–269. IEEE Computer Society, June 2005.

[31] M. Samuel, P. Saxena, and D. Song. Context-sensitive autosanitization in web templating languages using type qualifiers. In *Proceedings of the Conference on Computer and Communications Security*, Oct. 2011.

[32] B. Scholz, C. Zhang, and C. Cifuentes. User-input dependence analysis via graph reachability. Technical Report 2008-171, Sun Microsystems Labs, 2008.

[33] D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid. Abstracting symbolic execution with string analysis. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 13–22, 2007.

[34] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. Autoises: Automatically inferring security specification and detecting violations. In *USENIX Security Symposium*, 2008.

[35] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of Web applications. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2009.

[36] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. A systematic analysis of XSS sanitization in Web application frameworks. In *Proceedings of the European Symposium on Research in Computer Security*, Sept. 2011.