Practical Context-Aware Permission Control for Hybrid Mobile Applications

Kapil Singh

IBM T.J. Watson Research Center
kapil@us.ibm.com

Abstract. The rapid growth of mobile computing has resulted in the development of new programming paradigms for quick and easy development of mobile applications. Hybrid frameworks, such as PhoneGap, allow the use of web technologies for development of applications with native access to device's resources. These untrusted third-party applications desire access to user's data and device's resources, leaving the content vulnerable to accidental or malicious leaks by the applications. The hybrid frameworks present new opportunities to enhance the security of mobile platforms by providing an application-layer runtime for controlling an application's behavior.

In this work, we present a practical design of a novel framework, named MobileIFC, for building privacy-preserving hybrid applications for mobile platforms. We use information flow models to control what untrusted applications can do with the information they receive. We utilize the framework to develop a fine-grained, context-sensitive permission model that enables users and application developers to specify rich policies. We show the viability of our design by means of a framework prototype. The usability of the framework and the permission model is further evaluated by developing sample applications using the framework APIs. Our evaluation and experience suggests that MobileIFC provides a practical and performant security solution for hybrid mobile applications.

1 Introduction

With the development of new mobile platforms, such as Android and iOS, mobile computing has shown exponential growth in popularity in recent years. A major factor driving this growth is the availability of a huge application market that provides rich functionality ranging from banking to gaming to social networking. To benefit from the availability of a constantly growing consumer base, new services and applications are being built from the composition of existing ones at breakneck speed.

Most mobile operating systems currently use a capability-based permission system that mediates applications' access to device resources (such as camera) or user's data (such as contact lists). The operating system vary in the way the permissions are granted. For example, users approve the permissions at install time in Android while such approval is done at the time of first use in iOS.

The permission model, in the current form, suffers from two major limitations. First, the model is too coarse-grained and lacks flexibility to support rich security policies. For example, it does not allow conditional policies, such as location-based policies, to control permissions. Moreover, the permissions cannot be modified at runtime[1] and requires an explicit reinstallation of the application to include any changes. Second, the permission model only provides access control over the device resources by explicitly releasing corresponding capabilities to the applications. However, access control policies are not sufficient in enforcing the privacy of an individual: once an application is permitted access to a data or a resource, it can freely leak this information anytime to an external entity for personal gains.

To further facilitate quick application development, new programming frameworks have emerged to allow web technologies to be used as building blocks for native mobile applications. Such frameworks, such as PhoneGap [10], Sencha [11] and Worklight [5], enable automatic portability of the application onto multiple mobile platforms, such as Android, iOS, Blackberry, etc. A wide variety of such *hybrid* applications have been developed using these frameworks including some recent popular applications, such as BBC's Olympic coverage application [2] and IGN's mobile social network Dominate [6]. The hybrid application market is "on a hypergrowth trajectory" and is expected to continue its upward growth with the entry of new major players into the market [23].

While these platforms are known to provide benefits of portability and easier development, their usefulness to security has not been fully understood. In essence, they provide an interpretation layer or middleware where flexible security policies and enforcement mechanisms can be realized to control applications' access to device resources. The resources include personal user data such as contact list, and the content generated by the use of device sensors such as camera or GPS. The biggest advantage of hooking any security solution into this layer is that it does not require any support from or changes to the underlying operating system and the solution is readily portable to multiple mobile platforms.

In this work, we are concerned with protecting the user content from leaks by untrusted (malicious or vulnerable) *hybrid* mobile applications. We propose and implement a new framework, called *MobileIFC* (Mobile Information Flow Control), that leverages the mediation layer of the hybrid platform to support *runtime* enforcement of *fine-grained*, *context-driven* policies. MobileIFC allows the user to provide mandatory security policies for protection of his content, while at the same time enabling mobile applications to be more specific about their permission requirements. For example, the user can specify context-driven policies such as "Camera pictures taken at work should only be shared with company's servers". The applications can also specify finer-grained permission requirements such as "Camera pictures are only shared with Picasa".

To enable context-aware policies, MobileIFC resolves the context of the device and/or the application at runtime when resource access is requested by the

[1] iOS 5+ enables control over certain permissions, such as contacts and geolocation, after an application is installed.

application and permissions are subsequently adapted based on the resolved context. For location-driven policies as an example, MobileIFC taps into the geolocation API of the hybrid platform to resolve the location of the device before deriving the associated security policies.

This paper makes the following contributions:

- We address the challenge of protecting user's mobile data in the fast growing hybrid application market. In contrast to the existing security solutions that rely on OS modifications, our solution is realized at the application layer as an extension to the hybrid frameworks and hence is readily portable to multiple mobile platforms. To the best of our knowledge, we are the first to provide a comprehensive permission framework for hybrid applications.
- We propose a rich permission model that enables applications and users to specify fine-grained, context-aware policies.
- To show the viability of our design and enable rich policy enforcement, we develop a novel framework, called MobileIFC, that redesigns applications to support effective information flow control for hybrid applications and enables context-dependent policy resolution at runtime. We illustrate the applicability of MobileIFC by developing representative (banking, healthcare and financial management) applications on top of the framework and analyzing its performance and integration overheads.

2 Overview

MobileIFC is an architectural framework for executing hybrid mobile applications that enables users to share their private mobile content with *untrusted* applications. The framework, in turn, prevents these applications from leaking users' sensitive content. MobileIFC effectively provides complete mediation for all communication to and from these applications at runtime to enable users to administer fine-grained, context-aware policies that satisfy their privacy requirements.

Typical mobile applications leverage services rendered by other applications on the device and by network servers. As a result, they need to communicate with entities outside the MobileIFC system, called *external entities*, to perform specific tasks. For example, a social networking application may communicate with `www.cnn.com` to receive a daily news feed for the user. Additionally, it may seek the device's camera application to click and post the user's picture on his profile.

Currently, applications are more-or-less monolithically installed on the mobile OS and isolated from each other and from the underlying OS by default. The OS controls access to security-sensitive device resources such as Internet access. However, such access follows an all-or-nothing permission approach and does not support restricting Internet access to only specific external entities. Moreover, applications can also define their own permissions to control access to sensitive interfaces that they expose to other applications. The application-centric permission model is not sufficient for transitive policy enforcement allowing privilege escalation attacks as shown by the recent attacks [13, 15, 19].

Even after the current permission model is extended to make it fine-grained, access control, by itself, is not sufficient as it does not satisfy the principle of least privilege: even if an approved external entity, e.g. `www.news.com`, requires no user's personal information, the application can (mistakenly or maliciously) share with the external entity any piece of user information available to the application.

In the hybrid design, applications are hosted by the hybrid programming platform that provides a set of APIs to expose the functionality available to native applications. The platform itself along with the hosted hybrid application is deployed on the underlying OS as a native application. The platform requests the desired access or permissions from the mobile OS using the permission model supported by the OS. This makes the platform an ideal location to hook a reference monitor that controls all its granted permissions. As a result, it can selectively grant or revoke a subset of these permissions to the hybrid application based on finer-grained, context-aware policies.

The uniqueness of MobileIFC's design is attributed to techniques that enable efficient information flow control within the framework, thus allowing it to enforce fine-grained policies. We adapt some of the concepts from previous work in the social networking domain [27] to build MobileIFC suitable for the hybrid application environment. Information flow control in MobileIFC is enforced by design, i.e., MobileIFC redesigns the applications in order to achieve effective and efficient information flow control. The applications are split into a set of *chunks*[2]; a chunk being the smallest granularity of application code on which policies are administered by MobileIFC. A chunk is chosen based on what information the chunk has access to and what external entity it is allowed to communicate with.

From an end user's perspective, the applications are monolithic as the user does not know about the chunks. At the time of adding a particular application, the user is presented with a manifest that states what piece of user's private or sensor data is needed by the application and which external entity will it be sharing this data with. For example, the social networking application's manifest would specify that it shares any pictures it takes using the device's camera with *only* the social network's server. Note that the application does not need to reveal that it communicates with `www.news.com` as no user information is being sent to `www.news.com`. The user can now make a more informed decision before adding the application.

In addition to the approval-based approach, MobileIFC also allows the user to define his own privacy policies as functions of user/device resources (as input), external entities (as output), and device or application context (as associated condition). For example, a user can specify that the device's camera should not be available to any application at work, thus revoking social networking application's camera access at user's work location. Such user scenarios are realistic in the real world as shown by a recent policy change at IBM regarding iPhone Siri's sharing of voice data with Apple's servers [12].

[2] We use the term *chunks* instead of components to differentiate from the component-based architecture in Android.

Section 3 provides a detailed description of our design and how MobileIFC ensures that only approved flows are allowed. In this section, we present our trust model (Section 2.1) and discuss how MobileIFC's permission model enables rich security policy specification using some representative examples (Section 2.2). We use Android as the mobile OS of choice for discussions, though our cross-platform solution for hybrid applications is independent of any OS. We also use open-sourced PhoneGap as our representative hybrid framework; the concepts and solutions developed in our work can be similarly applied to other frameworks.

2.1 Trust Relationships and Threat Model

In this work, we are concerned with securing a user's private information from leaks by malicious attackers. Consequently, our trust model is defined from an end-user perspective. Note that in our framework, a user represents both individuals seeking protection of their data and administrative entities, such as corporations, which administer data for their employees and clients.

There are multiple parties that are involved in distributing and consuming a user's private information. First, the hybrid framework provide the necessary enforcement for a user's privacy policies and therefore is trusted in our framework along with the underlying OS. Second, mobile applications that are developed by third parties are untrusted by default. We assume that such applications can either be developed by malicious attackers with the sole purpose of collecting users' sensitive information, or are benign yet vulnerable to exploits that could result in information leaks.

For an information leak to be considered successful, the sensitive information must be passed to an *unintended* external entity. In our design, we consider three classes of external entities based on their associated trust. All external entities are *untrusted* by default unless they are approved by the user for data sharing (Section 3). Once approved, the external entity is considered *semi-trusted*, i.e., it may receive only the sensitive information for which it is approved. A *trusted* entity is allowed to receive sensitive information and is furthermore expected to filter any sensitive content from its output before providing it to the application. In other words, a trusted entity must act as a declassifier.

Our work prevents information leak of the content provided to the untrusted third-party applications. It cannot prevent use of outside channels by the approved external entities to share information once such entities get access to the information. This also implies that we only consider leakage protection on the device (client) side in case of a multi-tier application.

2.2 Policy Specification in MobileIFC

In this section, we use a representative banking application to show how rich security policies can be defined and enforced in MobileIFC to prevent applications from leaking user content. The policies are expressed via *fine-grained*, *context-aware* permissions along with other (possibly organization-specific) mandatory policies and subsequently enforced at runtime by the MobileIFC framework.

Note that while the mandatory policies allow the users to enforces their own privacy requirements and potentially prevent malicious behavior, we anticipate

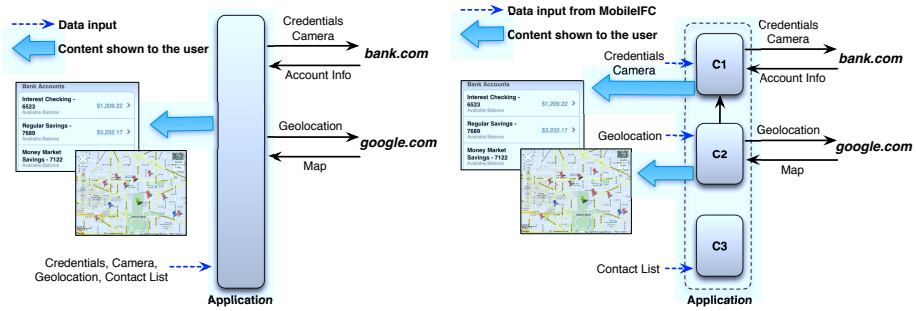

Fig. 1: Representative example of a banking application design for (a) current mobile applications and (b) MobileIFC.

that tradeoffs will arise: certain policy decisions that may prevent malicious application behavior may also disrupt the functionality of certain non-malicious applications. Such decisions must be made by users based on their specific organization's restrictions and requirements.

Representative Application: Banking. We use a banking application as our running example (Figure 1). The application takes a user's credentials to login into his bank account. The credentials are verified at the bank's servers before the account details are presented to the user. The banking application also communicates with third-party servers to present value-added services to the user, e.g., showing nearby bank locations using a map obtained from Google Maps. Moreover, it uses the device's camera to capture check images that are sent to the bank's servers. The application also accesses the contact list to facilitate selection of recipients for peer-to-peer (P2P) payments. The contact list information is not shared with any external entity.

The current Android permission model lists a set of pre-defined permissions that an application can request in order to access corresponding resources on the device. In our banking example, an Android application would need to request the `INTERNET` permission (to communicate with external entities) and `ACCESS_FINE_LOCATION` (to get access to user's geolocation to determine the closest bank locations) using a manifest. It would further request the `CAMERA` permission to have the capability to capture images with the device's camera and `READ_CONTACTS` permission to have access to the device's contact list. This manifest must be approved by the user before the application is installed.

We now give some examples of different types of security policies, and discuss how they can be accommodated in MobileIFC's permission model.

Information Flow Control with Functionality-Based Least Privileges This type of security property is concerned with protecting the user's private assets from leaks by untrusted applications. One security requirement for the banking application is that a user's bank credentials and location should be protected from eavesdropping or leakage. At the same time, the requirement should not break the application, i.e., the application should have enough privi-

leges to satisfy the desired functionality. This requirement leads to the following high-level security policies:

- The user's login credentials should only be shared with the bank's server `bank.com`.
- The device's geolocation information should only be shared with Google.

Limitations of the current model There are two major issues with the current access control model for Android applications. First, the resource access is coarse grained and does not follow the *principle of least privilege*. For the banking application, even if the application needs to communicate over the Internet only with its own server, it still possesses full capabilities to freely communicate information, such as the user's credentials, to any other external entities. Second, there is no correlation between specific data items and the external parties to which they are sent. As a result, there is nothing that prevents the application from sharing the user's banking credentials with Google.

Our permission model In our permission model, the application's manifest provides finer-grained requirements for its external communication. Specifically, it provides an input-to-output mapping, which represents what protected user/device information (asset) is to be shared with what external entity. For the banking application, this mapping would correspond to the set {(login credentials, `bank.com`), (geolocation, `google.com`)}. Our application design will ensure that the application conforms to the the requested (and approved) information flows (Section 3).

Context-Aware Security Properties This security property addresses conditional use of user content by the application. The conditions can be a derivative of the device state, such as the GPS location or time of the day. As an example of a situation where permissions depend on context, consider a scenario where an organization such as DoD wants to impose the requirement "No images should be captured at the Pentagon". This property maps to the following security policy:

- When the geolocation of the device corresponds to Pentagon's location coordinates, an application's camera capture ability should be disabled.

Limitation of the current model The current Android model does not consider any location-based permissions. Once the application has the `CAMERA` permission, it can freely capture pictures irrespective of the location.

Our permission model MobileIFC ensures that the camera is only activated when the device's geolocation is in a certain state. To address such a scenario, MobileIFC's design restricts the application to access the device's camera only through a prescribed API. MobileIFC's mediation layer resolves the required context to identify the device's current geolocation and then ensure that the camera is only activated in accordance with the policy under consideration.

3 MobileIFC Design

MobileIFC shifts the bulk of the performance costs of tracking information flows to the application development stage. Instead of using traditional taint

tracking mechanisms [17], MobileIFC exposes the security-relevant information flows within an application by redesigning the application. It splits the application into chunks that represent the smallest unit of flow tracking within the MobileIFC framework. A chunk represents a piece of code that is uniquely identified by its input values and the external entities it needs to communicate with. For instance in our representative banking example, chunk C_2 takes in geolocation as the input and communicates with `google.com` as the external entity (Figure 1(b)).

While an ideal application design in MobileIFC would follow the principle of least privilege, MobileIFC does not place any restriction on the developers on how to design their application. In other words, it means that the actual functionality, semantics, and runtime characteristics are not of interest in MobileIFC and are left to the developer. This provides the application developer with enough freedom and flexibility to build rich applications. However, MobileIFC ensures that only the flows approved by the user (or allowed by his mandatory policies) are allowed, thus forcing the application developers to make any intended communication explicit. For instance, a developer can design the banking application in two ways. First, he can follow the current monolithic application design as shown in Figure 1(a) and in that case, the application's manifest would declare that it requires user's credentials, camera, geolocation and contact list as input and `bank.com` and `google.com` as the external entities. It effectively means that the complete application would act as a single blackbox and any of the input parameters are allowed to be shared with any of the external entities. Note that even this first design is an improvement over existing application design as it explicitly enumerates the allowed external entities. Alternatively, he can design the application as shown in Figure 1(b). Since the second design splits the information flow from the input parameter to the external entity, each chunk possess lower privileges (and only privileges that it needs) thus reducing the attack surface in case of a malicious application or confining any exploit to within a chunk in case of a vulnerability. As a result, the user would be more inclined to approve the second design in comparison to the first.

We envision that an application can be automatically split into chunks, where a chunk boundary is effectively decided by individual user policies. Our current system relies on application developers to manually split the applications; we plan to develop an automated system for application splitting as future work.

3.1 Confinement of chunks

The chunks of an application encapsulate different levels of private information for the users. Therefore, these chunks need to be isolated from each other in order to prevent information leaks. Since hybrid applications use webview for all layout rendering, they are administered by the Same Origin Policy (SOP). However, since the application's HTML files are associated with the `file://` protocol, all pages have the same origin thus neutralizing any potential benefit of SOP. Moreover, cross-origin AJAX requests are enabled allowing the application chunks to freely communicate with any external entities.

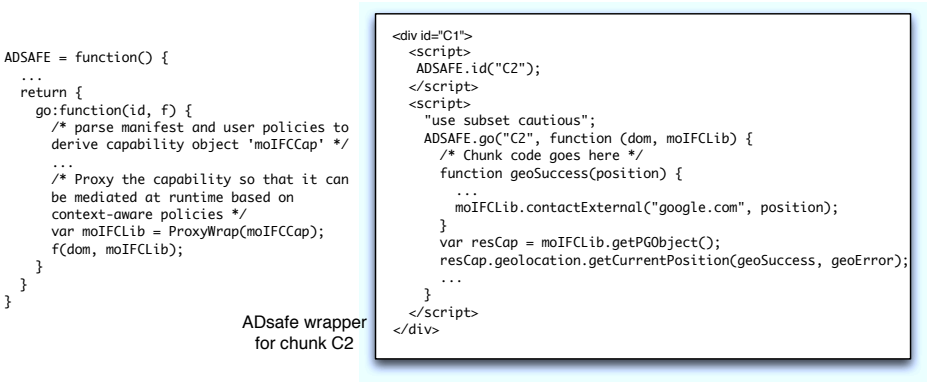

Fig. 2: ADsafe-based chunk confinement and monitoring in MobileIFC.

A script on a page has intimate access to all information and relationships of the page. As a result, the chunks are free to access the Document Object Model (DOM) objects of other chunks. Additionally, the chunks are allowed to access the device's resources using the APIs exposed by the hybrid platform. Therefore, any confinement mechanism should (1) constrain a chunk to access only its own DOM objects with no view of other chunks' objects, and (2) limit a chunk's access to only approved resources on the device.

In order to constrain chunks into their own control domain, we limit the application code to be written in an object capability language called ADsafe [1]. In an object capability language, references are represented by capabilities and objects are accessed using these references. ADsafe defines a subset of JavaScript that makes it safe to include guest code (such as third-party scripted advertising or widgets) on any web page. ADsafe removes features from JavaScript that are unsafe or grant uncontrolled access to elements on the page. Some of the features that are removed from JavaScript are global variables and functions such as `this`, `eval` and `prototype`. It is powerful enough to allow guest code to perform valuable interactions, while at the same time preventing malicious or accidental damage or intrusion.

To monitor and control access to the device's resources, we modified ADsafe to exclude any PhoneGap API calls that provide a direct handle to access the resources and to invoke their functionality. As an example, the API `navigator.camera` that is used to capture an image using the device's camera is banned. The access to provided indirectly by means of a chunk-specific wrapper object that exposes only a subset of the APIs as allowed by the approved permissions for the chunk (Figure 2).

3.2 Realization of security policies

We developed a proxy engine that mediates all calls to PhoneGap APIs and realizes the policy requirements of the user. The proxy engine takes as input any mandatory security policies specified by the user. Since the mediation is

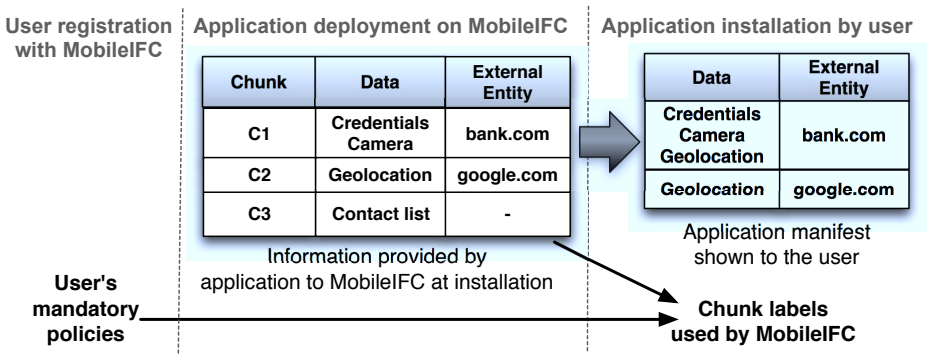

Fig. 3: Typical life cycle of an application in MobileIFC.

done at runtime (i.e. at the time of use), any runtime modifications to the user's mandatory policies are also incorporated (Figure 2).

The user policies dictate the book-keeping tasks taken up by the proxy engine. For context-aware policies (Section 2.2), the engine analyzes the input policy to resolve any unknown contexts before verifying them against the specified conditions. For conditional location-based policies as an example, it resolves user's current geolocation before checking the associated condition. Note that the proxy engine runs within the trust domain of the hybrid platform, so it is privileged with all the permissions that are associated with the platform, effectively enabling it to resolve contexts by utilizing the device's sensors.

The current design of MobileIFC maintains a mapping between permissions and the corresponding PhoneGap APIs that require these permissions. For example, CAMERA permission in Android corresponds to the `navigator.Camera` and `navigator.Capture` objects in PhoneGap. Each of these objects have multiple member properties and functions that administer certain ability to the picture capturing functionality. The permissions are specified in terms of the labels (e.g. CAMERA) that give permission to access a particular resource (e.g. device's camera).

Our design also supports finer-grained permission specification, i.e., at the level of specific APIs instead of specific resources. However, specifying such finer policies must be done sensibly, as it increases bookkeeping and needs better understanding of the APIs by the user, and therefore could potentially break existing interactions if policies are specified incorrectly.

3.3 Application Lifecycle in MobileIFC

Figure 3 shows a typical life cycle of an application. The user first registers with the MobileIFC framework by providing his mandatory privacy policies specific to his sensitive data and resources. For example, he can specify that his contact list should never be shared with any external entity. The developer of an application decides on the structure of the chunks for that application and during the application's deployment on MobileIFC, he specifies the information required by each chunk and the external entity a particular chunk needs to com-

municate with. MobileIFC uses this information to generate the manifest for the application. As shown in the figure, a manifest is basically a specification of the application's external communications (irrespective of the chunks) along with the user's data that is shared for each communication. This manifest needs to be approved by the user before the application is installed for the user. Additionally, the MobileIFC platform ensures that all of the application's chunks comply with the user's mandatory privacy policies and the manifest approved by the user. For any context-aware policies, the context is resolved at runtime and associated conditions are verified before any access is granted.

3.4 The Banking Application on MobileIFC

To illustrate the application design within MobileIFC, let us revisit our banking application introduced in Section 2.2. To satisfy the user's privacy requirements, two conditions should be fulfilled: (1) no banking data should be shared with Google; and (2) user's contact list should be kept private.

In the current application design, the application can freely leak any content it possesses to any external entity after it has the INTERNET permission. Even if the external entities are restricted to only `bank.com` and Google, the application would be able to pass all information about the user, including the details of his bank account and his check images, to Google (see Figure 1(a)). Moreover, his contact list can be shared with `bank.com`.

The division of an application into multiple chunks allows the application writer to develop different functionality within an application that relies on different pieces of the user information. In the MobileIFC framework, the banking application would be split into three chunks as shown in Figure 1(b). Chunk C_1 can only communicate with `bank.com` and has access to its login information (such as userid and password). Additionally, it also receives check images taken from the device's camera. Chunk C_2 has no access to any of the banking information and interacts with Google using the user's current geolocation to produce a map of the bank's locations nearest to the user. Chunk C_3 has access to user's contact list, but does not communicate with any external entity.

Since chunk C_2 is given access to user's geolocation information, this is the only information it can communicate to an external entity. Moreover, it is restricted to communicating only with Google. As per basic information flow-control rules, information can flow from a less restricted to a more a restricted chunk, thereby allowing one-way communication from C_2 to C_1. As a result, C_2 can pass a user's selected branch location on the map to C_1, which, in turn, uses the selection to show the local information of that branch. Since C_3 cannot communicate with any external entity, it cannot leak any information outside the MobileIFC framework. This enables C_3 to receive any information from other chunks as well as any additional user content such as the contact list.

In additional to the security benefits provided by MobileIFC, its design also supports graceful degradation to partial usability for the applications. Taking the case of our banking application, a user can decide not to share his geolocation with Google by not approving that part of the manifest. This would not impact

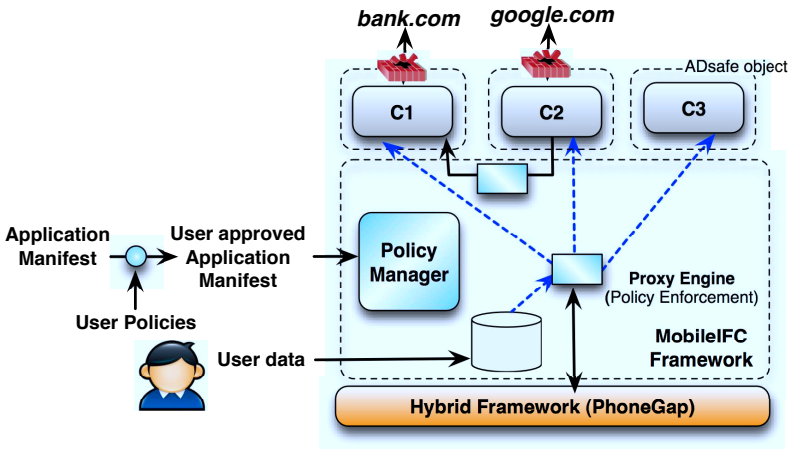

Fig. 4: High-level view of MobileIFC implementation.

the core banking functionality of the application and if designed for graceful degradation, it would only partially impact the overall user experience.

4 Implementation

One of the goals of our implementation is to require minimum changes to the mobile user experience and minimum efforts from the application developers. From the user's perspective, the only new requirement of MobileIFC is to attach privacy policies to his sensitive data and device's resources. If the user opts not to provide such mandatory policies (before application installation and/or at runtime), MobileIFC still defaults to the install time-approval model even though it can be more fine-grained than the current permission models. For application developers, the additional effort means that the application has to be structured into chunks along security-relevant boundaries, instead of strict functionality boundaries.

In view of the aforementioned goals, MobileIFC's implementation complements the PhoneGap framework to include several new features and functionality. First, it provides an interface for users to specify their fine-grained, context-aware privacy policies and also enable them to modify these policies even after application installation. The policies can be made applicable to one or more applications. Second, the implementation extends the support for application manifests by enabling application to include fine-grained requirements. Note that the extended manifest file is parsed by MobileIFC and not by the underlying OS and hence no changes are needed in the OS. Third, it provides tools to refine and merge user policies and application manifests. Finally, it provides the platform for application deployment that efficiently deploy the chunks, associate appropriate information flow labels to each chunk based on the user policies and provides the enforcement layer to provably ensure that communication patterns of the application always satisfy the chunk labels. The platform also resolves

```
<?xml version="1.0" encoding="utf-8"?>
<policy>
   <condition name="worklocation">
      <type value="geolocation"></type>
         <latitude>35.769915</latitude>
         <longitude>-78.599146</longitude>
   </condition>
   <permission name="permission.CAMERA"
          condition="worklocation" condition-match="deny" />
</policy>
```

Fig. 5: Context-aware policy example in MobileIFC.

context, such as the device's location, for administering context-aware policies by invoking appropriate resource access APIs of the underlying OS.

Figure 4 shows a high level view of our implementation presented in regards to our running banking example. The application chunks are contained and deployed as individual ADsafe objects to achieve complete isolation between chunks and to prevent any direct access to the device's resources. MobileIFC provides a set of APIs that are exposed to the application chunks to (1) access resources and (2) support both unidirectional and bidirectional communication among the chunks. These APIs are available as an add-on library for the application developers as part of the software development process (e.g. as an eclipse add-on) and packaged into the PhoneGap framework to be made available to the application code at runtime. We anticipate that packaging of the application with the hybrid framework would be done by a trusted party, such as an app store, to prevent malicious application developers to deploy a modified hybrid framework.

During the application's deployment into the app store, the application developers provide their chunk requirements as part of a manifest file. For our implementation, the manifest's specification is build on top of Android's manifest format to include conditions for specifying fine-grained requirements. For policy specification, we currently provide our own custom language for writing the privacy policies (see Figure 5 for an example), however, we are in the process of porting the standard policy language, XACML [28], to specify such policies. The user can specify his privacy policies in the language using the interfaces provided by MobileIFC.

At application installation, MobileIFC verifies whether the application requirements detailed in the manifest satisfy the user policies and informs the user in case of conflicts. If the user policies are not marked as mandatory, the user has the option to resolve the conflicts before the application is added. At the time of approval, the user can selectively choose to prevent certain flows at the cost of degradation of functionality. The approved flows of the user manifest are fed to the *Policy Manager*, which applies the mediation policies into the *Proxy Engine* based on the manifest. The users can also modify their policies using MobileIFC's interfaces any time after the application's installation with the updates being handled by the Policy Manager.

The Policy Manager translates the high-level user policies into low-level, pluggable deployment of such policies. It creates *templates* for the policies, where context-based conditions are specified as informative variables that need to be resolved by the Proxy Engine at runtime. In a simplistic representation, the state-based policy from Section 2.2 would translate into the following:

```
if VAR(geolocation.getCurrentLocation) == CONST(Pentagon)
            !allow Permissions.CAMERA
```

This directs the Proxy Engine to resolve the `VAR` by invoking the PhoneGap API `geolocation.getCurrentLocation` and compare it with the `CONST Pentagon` that is supplied as part of the high-level policy. The condition is verified before access to any API that requires CAMERA permission is provided.

The MobileIFC framework tracks and enforces information flow using a labeling system based on existing models [24, 30]; we omit further details in the paper.

5 Evaluation

The main goals for our evaluation are to determine whether the user's privacy policies are actually enforced for an application deployed on MobileIFC and whether the impact this architecture has on the mobile user and on the application developer is acceptable. To determine whether the policy enforcement in MobileIFC protects the user's privacy, we modified our banking application such that in addition to its normal functionality, it would also try to leak information by creating different attack scenarios. For example, the application would try to send the bank credentials to `google.com`. The privacy policies we considered in our evaluation restricted the communication of banking credentials only to `bank.com`, thus these information leaks have to be stopped by MobileIFC. To determine whether MobileIFC is an attractive approach for the end user, we analyzed the performance impact of its runtime enforcement. Finally, to determine the impact on the application developer, we analyzed the burden on the development process by measuring the amount of code changes necessary to adapt the application to the MobileIFC platform. In addition to the banking application, we also developed a healthcare application (based on Microsoft's Health Vault [7]) and a financial management application (based on `mint.com` [8]) to show the viability of application development in MobileIFC.

5.1 Security Analysis

Our analysis aims to show that MobileIFC prevents applications from leaking any user information. We tested the ability of our prototype by creating synthetic exploits that attempt to break out of MobileIFC's information flow control model to leak user information. We enhanced the ability of our banking application to launch these attacks against our prototype; if successful, these attacks would allow the application to leak information to entities outside the system.

Table 1 shows the results of testing our prototype against a wide range of these synthetic attacks. In all our experimental tests, MobileIFC successfully pre-

Attack	Attack Step	Example attack in the banking application	Prevented by MobileIFC?
A1	One chunk creating illicit connection to another chunk	C3 makes a connection to C2	√
A2	Leaks via the reverse path of a unidirectional inter-chunk communication	C1 leaking credentials to C2	√
A3	Chunk retrieves unapproved user information	C2 retrieves contact list	√
A4	Leaks to an unknown external entity	C3 leaks contact list to `evil.com`	√
A5	Leaking restricted information to an allowed external entity	C1 sends credentials to `google.com`	√

Table 1: Prevention of information leaks against various synthetic attacks.

vented all leaks before the information could be passed outside the system. Our ADsafe-based containment of chunks and complete mediation of communication to external entities by MobileIFC contributed to the prevention of A1 and A4. A2 was prevented by the one-way communication enforcement of MobileIFC. All access to user data is administered by MobileIFC thus preventing A3. Finally, the approved external entity for a chunk also determines the input information it can receive (either from MobileIFC or another chunk). As a result, attack A5 is implicitly prevented at chunk creation.

5.2 Integration Overhead

An application developer tasked with developing hybrid applications for MobileIFC faces two challenges. First, the application code must be structured into chunks and, second, the chunks need to be adapted to use MobileIFC's APIs for accessing data and resources, or to communicate with each other. The restructuring challenge is tackled to a large degree by existing software development methods that engineer the code into reusable and maintainable modules. In other words, current software engineering practices would naturally lead to the formation of natural chunks within the application code. While these chunks are defined along functional lines (i.e., they reflect self-contained, inter-related code and data elements), it is highly probable that they would serve as chunks in MobileIFC, which defines chunks based on the communication requirements with external entities.

The second challenge, of adapting chunks to use MobileIFC's APIs, requires understanding of the APIs on the part of the developer. While we preserve the signature of the APIs for data/resource access from the original PhoneGap APIs, we introduce new APIs for uni- and bi-directional communications. We designed the MobileIFC support library to minimize the complexity of code changes required by an application, as shown in the example below.

In a monolithic design, after the application receives the user's selected bank location on the map, it makes the following procedure call:

```
setSelectedLocation(bankLocationID);
```

In MobileIFC design, this call would be in the form of a inter-chunk unidirectional call from C_2 to C_1 as follows:

```
MobileIFC.callRemoteFunctionNoReturn
   ("C1", "setSelectedLocation", bankLocationID);
```

While this code transformation is currently done manually, the simplicity of the change and its purely syntactic form means that it can be automated, possibly as part of the software development environment.

While MobileIFC requires additional effort from the application developers (to compensate for effective enforcement benefits at runtime), our experience developing the three representative (banking, healthcare and financial management) applications show that this effort is reasonably low and can be further reduced by automating the chunking process.

5.3 Performance Estimates

With an new architectural framework and a new way of developing applications, it is difficult to accurately predict the impact of our design on the performance of these applications. Most of the cost to provide information flow control is amortized at application initialization as each chunk is only given access to the capability object of the resources that are allowed for that chunk (Figure 2). This object is modified accordingly to include any runtime policy changes. This is sufficient for flow control if no context-aware policies are specified for a resource.

In cases where context-aware policies are defined, the context needs to be resolved at runtime at the time when resource access is requested. This results in runtime performance overhead associated with mediation of resource access and resolution of context. To get a rough estimate of the cost of supporting the MobileIFC design and the overhead involved in our system, we conducted experiments against our sample banking application, measuring overhead imposed by the mediating design of MobileIFC.

The experiments were performed on Motorola Atrix phone with dual-core 1GHz processor and 1 GB RAM running Android 2.3.4. Each test was run 10 times and values were averaged. The results show that the overhead introduced by MobileIFC's mediated checks is negligible with a each check amounting to 5.2ms. The cost of context resolution was dependent on the sensor being queried, with values of 1.3 seconds for geolocation resolution, 3.5 seconds for access point lookups and 5.2 seconds for Bluetooth device discovery.

While these performance numbers may vary considerably based on the hardware sensors available in the mobile device, they still provide an intuition that the user's runtime experience of the application would potentially be impacted by context resolution. These numbers can be amortized by caching the results

of sensor queries across applications and by intelligent sampling. We plan to consider such options as part of our future work.

6 Discussion

In this section, we discuss limitations of the application design in MobileIFC and address some of the challenges originating from the new requirements imposed by our design.

MobileIFC's containment mechanism uses ADsafe to limit access of the application code to within chunk boundaries. ADsafe only applies to web technologies that are primarily used to develop hybrid applications. However, certain hybrid frameworks such as PhoneGap also support an ability to add plugin code in the native programming language of the underlying OS (e.g. Java for Android and Objective-C for iOS). Such code also needs to be constrained to control access to the APIs exposed by the OS. There are multiple approaches to address this challenge. The plugin code inherits the permissions given to the hybrid framework and therefore, the first approach is to limit the permissions given to the hybrid platform that would also constrain the plugin. However, support of a new permission model would need modifications to the underlying OS. The second approach would be to limit the plugin to use safe subsets of the plugin's programming language (such as Joe-E for Java [20]). Once the plugin code is constrained, mediation similar to MobileIFC can be applied to enforce specific policies. We plan to evaluate some of these approaches as part of future work.

In the current MobileIFC implementation, the application developers are vested with the additional responsibility to partition their applications along security-relevant boundaries. MobileIFC's design, of only allowing flows that are approved, ensures that an application cannot cheat about its requirements. From the application developer's perspective, our design has the additional benefit of isolating bugs or vulnerabilities within a chunk, giving them another incentive to adopt MobileIFC. As part of our future work, we plan to automate the process of creating logical boundaries within existing applications in order to partition them into chunks based on their input and output requirements. We will explore ways to leverage source and binary analysis techniques to partition the applications, thereby reducing the burden on the application developers, while at the same time preserving the privacy guarantees. Such solutions can be integrated into development tools such as Worklight Studio [5] to facilitate application development for MobileIFC.

While our design goal is to limit the burden on the users, MobileIFC does impose new usability requirements. The users need to understand the risk associated with sharing their data with various external entities and formulate appropriate policies as per their individual requirements. While corporate administrators can be expected to be better informed and to develop suitable policies for corporate users, regular users can use external resources such as Norton Safe Web [9] to make trust decisions about external entities. Moreover, our policy language is simple (Figure 5) and can be further complimented by a usable interface for improved usability.

7 Related Work

Mobile application security has been a major research focus in recent years. Research has analyzed the security issues of mobile applications for different mobile platforms, mostly focused on Android [17, 19, 31] with some work targeting iOS [16]. These works mostly target offline analysis of mobile applications looking for malicious behavior [31], or security evaluation of mobile platforms and their permission models [18, 19]. Other research target runtime analysis of the applications and the underlying platforms [13, 17].

TaintDroid [17] is one of the first systems to address IFC for mobile platforms. TaintDroid exploits dynamic taint analysis in order to label privately declared data with a taint mark, audit on-track tainted data as it propagates through the system, and warn the user if tainted data aims to leave the system at a taint sink (e.g., network interface). However, TaintDroid is limited in its tracking of control flows due to high performance penalties. AppFence [21] is another system that extends the TaintDroid framework by allowing users to enable privacy control mechanisms to help difference between authorized data sharing and malicious data leakage. While MobileIFC shares a common goal of detecting unauthorized leakage of sensitive data, its approach is orthogonal to the one taken by TaintDroid. Since it pushes the bulk of design decisions before runtime and does not require low-level taint tracking, MobileIFC successfully improves efficiency and simplifies enforcement at runtime. Moreover, we are addressing the IFC for hybrid applications and hence MobileIFC's IFC does not require any changed to the underlying operating system. To the best of our knowledge, we are the first to provide an IFC solution for hybrid applications.

Saint [26] introduces a fine-grained access control model that enforces security decisions based on signatures, configurations and contexts (e.g., phone state or location). Saint relies on on application developers to define security policies, therefore, it suffers from the issue of malicious applications intentionally leaking user data. By contrast, MobileIFC's permission model is user-centric and protects against both vulnerable and malicious applications. Moreover, we believe that users are better suited to understand the value of their own personal data or resources. As previously mentioned, users also include system administrators of corporations, therefore MobileIFC also enables enforcement of corporate security policies in BYOD setups.

Both Apex [25] and CRePE [14] focus on enabling/disabling functionalities and enforcing runtime constraints on mobile applications. While Apex provides the user with the means to selectively choose the permissions and runtime constraints each application has, CRePE enables the enforcement of context-related policies similar to MobileIFC. However, their enforcement is too coarse-grained and is limited to only access control. For instance, networking would be disabled for all applications, not just particular ones. Moreover, it requires rooting of the device for enable enforcement in the Android OS, while our solution provides the enforcement in the application's hybrid runtime. Aurasium [29] and Dr. Android [22] use application repackaging to enable policy enforcement at runtime and does not require any OS modifications. Even though both systems support

finer-grained policies, such as allowing access to specific external IPs, they still do not provide information flow control. However, MobileIFC can benefit from some of these repackaging techniques to automatically modularize applications into chunks. We will explore this as future work.

New mobile OSes, such as ChromeOS [3] and FirefoxOS [4], enable web applications to have native access to device's resources. These new platforms provide alternatives to the traditional mobile OSes (such as Android and iOS), and require explicit installation. In contrast, hybrid platforms enable web technologies to be used for application development in traditional OSes. While our current solution is built for hybrid platforms, some of the techniques, such as context-aware permission control, can be applied to the new OSes; one difference being that MobileIFC has to be built into the OS itself.

8 Conclusions

We presented a practical design of a novel framework, called MobileIFC, that considerably improves privacy control in the presence of untrusted hybrid mobile applications. Our design allows the applications to access sensitive user data while preventing them from leaking such data to external entities. MobileIFC redesigns the applications to achieve efficient information flow control over user content passed through these applications.

We also introduced a flexible permission model that enables the users to specify fine-grained, context-aware policies. Our model supplements user approved policies with an ability to specify generic, high-level, mandatory policies. We developed a working prototype of our MobileIFC system and used it for developing representative applications to demonstrate viability of MobileIFC and its applicability to real-world scenarios.

With portability and ease of application development driving the evolution of new hybrid frameworks, the number of hybrid applications will continue to rise. With their increased reliance on new code (via JavaScript) available at runtime, hybrid applications will stretch the limits of the current solutions to mobile application security. We believe that MobileIFC provides a practical direction for the development of efficient security and privacy solutions for mobile applications.

References

1. ADSafe. http://www.adsafe.org.
2. Apps Created with PhoneGap. http://phonegap.com/app/.
3. Chrome OS. http://www.chromium.org/chromium-os.
4. Firefox OS. https://developer.mozilla.org/Firefox_OS.
5. IBM Worklight. http://www-03.ibm.com/software/products/us/en/worklight/.
6. IGN Dominate. http://wireless.ign.com/articles/116/1167824p1.html.
7. Microsoft HealthVault. http://www.microsoft.com/en-us/healthvault/.
8. Mint. https://www.mint.com/.
9. Norton Safe Web. http://safeweb.norton.com/.
10. PhoneGap. http://www.phonegap.com.

11. Sencha. http://www.sencha.com.
12. B. Bergstein. IBM Faces the Perils of "Bring Your Own Device", May 2012. http://www.technologyreview.com/news/427790/ibm-faces-the-perils-of-bring-your-own-device/.
13. S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *NDSS*, San Diego, CA, Feb. 2012.
14. M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-related Policy Enforcement for Android. In *ISC*, Boca Raton, FL, Oct. 2011.
15. L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege Escalation Attacks on Android. In *ISC*, Boca Raton, FL, Oct. 2011.
16. M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *NDSS*, San Diego, CA, Feb. 2011.
17. 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 *OSDI*, Vancouver, Canada, Oct. 2010.
18. W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *CCS*, Chicago, IL, Nov. 2009.
19. A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-Delegation: Attacks and Defenses. In *USENIX Security Symposium*, San Fransisco, CA, Aug. 2011.
20. M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable Functional Purity in Java. In *CCS*, Alexandria, VA, Oct. 2008.
21. P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. "These Aren't the Droids You're Looking For": Retrofitting Android to Protect Data from Imperious Applications. In *CCS*, Chicago, IL, Oct. 2011.
22. J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *SPSM Workshop*, Raleigh, NC, Oct. 2012.
23. P. McDougall. IBM Acquires Mobile Specialist Worklight. http://www.informationweek.com/news/development/mobility/232500829.
24. A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *SOSP*, Saint Malo, France, Oct. 1997.
25. M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *ASIACCS*, Beijing, China, Apr. 2010.
26. M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *ACSAC*, Honolulu, HI, Dec. 2009.
27. K. Singh, S. Bhola, and W. Lee. xBook: Redesigning Privacy Control in Social Networking Platforms. In *USENIX Security Symposium*, Montreal, Canada, Aug. 2009.
28. M. Verma. XML Security: Control information access with XACML. http://www.ibm.com/developerworks/xml/library/x-xacml/.
29. R. Xu, H. Sadi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *USENIX Security Symposium*, Bellevue, WA, Aug. 2012.
30. N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *OSDI*, Seattle, WA, November 2006.
31. Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE S&P*, San Fransisco, CA, May 2012.