Semantically Rich Application-Centric Security in Android

Machigar Ongtang, Stephen McLaughlin, William Enck and Patrick McDaniel
Department of Computer Science and Engineering
The Pennsylvania State University, University Park, PA 16802
Email: {ongtang,smclaugh,enck,mcdaniel}@cse.psu.edu

Abstract—Smartphones are now ubiquitous. However, the security requirements of these relatively new systems and the applications they support are still being understood. As a result, the security infrastructure available in current smartphone operating systems is largely underdeveloped. In this paper, we consider the security requirements of smartphone applications and augment the existing Android operating system with a framework to meet them. We present *Secure Application INTeraction (Saint)*, a modified infrastructure that governs install-time permission assignment and their run-time use as dictated by application provider policy. An in-depth description of the semantics of application policy is presented. The architecture and technical detail of Saint is given, and areas for extension, optimization, and improvement explored. As we show through concrete example, Saint provides necessary utility for applications to assert and control the security decisions on the platform.

Keywords-mobile phone security; Android; application interactions; mediation;

I. INTRODUCTION

Smartphones have spurred a renaissance in mobile computing. The applications running on smartphones support vast new markets in communication, entertainment, and commerce. Hardware, access, and software supporting such applications are now widely available and often surprisingly inexpensive, e.g., Apple's App Store [1], Android's Market [2], and BlackBerry App World [3]. As a result, smartphone systems have become pervasive.

Mobile phone applications are shifting from stand-alone designs to a collaborative (service) model. In this emerging environment, applications expose selected internal features to other applications, and use those provided by others. In the latter case, applications simply search and use appropriate providers of a service type at run-time, rather than bind itself to specific implementations during development. This allows a rich culture of "use and extend" development that has led to an explosion of innovative applications. This culture is possibly best illustrated in the Android[1] operating system community.

The security model of the Android system (and that of many other phone operating systems) is "system-centric". Applications statically identify the permissions

[1]http://www.android.com

that govern the rights to their data and interfaces at installation time. However, the application/developer has limited ability thereafter to govern to whom those rights are given or how they are later exercised. In essence, permissions are asserted as often vague suggestions on what kinds of protections the application desires. The application must take on faith that the operating system and user make good choices about which applications to give those permissions—which in many cases is impossible because they do not have sufficient context to do so.

Consider a hypothetical PayPal service built on Android. Applications such as browsers, email clients, software marketplaces, music players, etc. use the PayPal service to purchase goods. The PayPal service in this case is an application that asserts permissions that must be granted to the other applications that use its interfaces. What is a legitimate application? Only PayPal the application (really PayPal the corporation) is in a position to know the answer to that question. This is more than simply a question of who is making the request (which in many cases in Android is itself unknowable), but also where, when, how, and under what conditions the request is being made. Unfortunately, Android does not provide any means for answering those questions or enforcing a security policy based upon them. Simply put, *the Android system protects the phone from malicious applications, but provides severely limited infrastructure for applications to protect themselves*. Based on extensive development of Android applications, we observe three essential application policies not available to applications in the Android security framework:

1) *Permission assignment policy* - Applications have limited ability to control to whom permissions for accessing their interfaces are granted, e.g., white or black-list applications.
2) *Interface exposure policy* - Android provides only rudimentary facilities for applications to control how their interfaces are used by other applications.
3) *Interface use policy* - Applications have limited means of selecting, at run-time, which application's interfaces they use.

This paper introduces the *Secure Application INTeraction (Saint)* framework that extends the existing Android security architecture with policies that address these key application requirements. In the Saint-enhanced infrastructure,
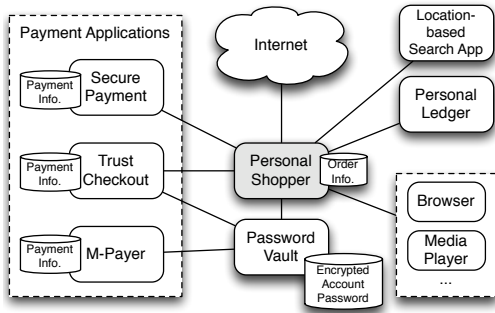
Figure 1. The PersonalShopper application finds desired items at the discretion of the user and interacts with vendors and payment applications to purchase them.

applications provide installation-time policies that regulate the assignment of permissions that protect their interfaces. At run-time, access of or communication between applications is subject to security policies asserted by both the caller and callee applications. Saint policies go far beyond the static permission checks currently available in Android by restricting access based on run-time state, e.g., location, time, phone or network configuration, etc. We define the Saint framework and discuss the complexities of augmenting Android with extended policy enforcement features, and develop mechanisms for detecting incompatibilities and dependencies between applications. We begin our discussion with a motivating example.

II. SMARTPHONE APPLICATION SECURITY

Figure 1 presents the fictitious PersonalShopper smartphone shopping application. PersonalShopper tracks the items a user wishes to buy and interacts with payment applications to purchase them. A user enters desired items through the phone's user interface (potentially by clicking on items on a browser, media player, etc.), creating a vender independent "shopping cart". Users subsequently acquire items in one of two ways. The user can direct the application to "find" an item by clicking on it. In this case the application will search known online vendors or shopping search sites (e.g., Google Product Search) to find the desired item. Where multiple vendors provide the same item, the user selects their vendor choice through a menu. The second means for finding a product is by geography—a user moving through, for example, a mall can be alerted to the presence of items available in a physical store by a location-based search application. In this case, she will be directed to the brick-and-mortar vendor to obtain the item.

Regardless of how the item is found, PersonalShopper's second objective is to facilitate the purchase process itself. In this case, it works with our example checkout applications SecurePayer and TrustCheckout. PersonalShopper accesses checkout applications and acts as an intermediary between the buyer and the merchants to both improve the efficiency of shopping and to protect customer privacy. The application and the services they

use will interact with password vaults to provide authenticating credentials. Following their completion, the transactions are recorded in a personal ledger application.

Consider a few (of many) security requirements this application suggests:

1) PersonalShopper should only use trusted payment services. In Figure 1, it may trust SecurePayer and TrustCheckout, but does not trust other unknown payment providers (e.g., the M-Payer provider).
2) PersonalShopper may only want to restrict the use of the service to only trusted networks under safe conditions. For example, it may wish to disable searches while the phone is roaming or highly exposed areas (e.g., airports) or while battery is low.
3) PersonalShopper may require certain versions of service software be used. For example, the password vault application v. 1.1 may contain a bug that leaks password information. Thus, the application would require the password vault be v. 1.2 or higher.
4) PersonalShopper may wish to ensure transaction information is not leaked by the phone's ledger application. Thus, the application wishes to only use ledgers that don't have access to the Internet.
5) Security requirements may be placed on PersonalShopper by the applications and services it uses. For example, to preserve location privacy, the location-based search application may only provide PersonalShopper location information only where PersonalShopper holds the permissions to access location information itself, e.g., the phone's GPS service.

None of these policies are supported by the current Android security system. While some of these may be partially emulated using combinations of complex application code, code signing, and permission structures, they are simply outside the scope of Android's security policy. As a consequence (and core to our extensive experience building systems in Android), applications must cobble together custom security features on top of the rudimentary structures currently provided by the Android system. Where possible at all, this process is ad hoc, error prone, repetitive, and inexact.

What is needed is for Android to provide applications a more semantically rich policy infrastructure. We begin our investigation by outlining the Android system and security mechanisms. Section IV examines a spectrum of policies that are potentially needed to fulfill the applications' security requirements, highlighting those cannot be satisfied by the current Android. We then introduce goals, design, and implementation of the Saint system.

III. ANDROID

Android is an mobile phone platform developed by the Google-led Open Handset Alliance (OHA).[2] The platform

---

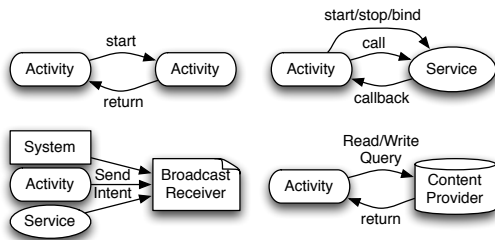[2]http://www.openhandsetalliance.com/

Figure 2. Typical Android application component IPC

quickly became popular amongst the developer community for its open source nature and adoption by telecommunications providers world-wide. While Android is based on Linux, the middleware presented to application developers hides traditional OS abstractions. The platform itself focuses on applications, and much of the core phone functionality is implemented as applications in the same fashion used by third-party developers.

Android applications are primarily written in Java and compiled into a custom byte-code (DEX). Each application executes in a separate Dalvik virtual machine interpreter instance running as a unique user identity. From the perspective of the underlying Linux system, applications are ostensibly isolated. This design minimizes the effects of a compromise, e.g., an exploited buffer overflow is restricted to the application and its data [4].

All inter-application communication passes through middleware's *binder* IPC mechanism (our discussion assumes all IPC is binder IPC). Binder provides base functionality for application execution. Applications are comprised of *components*. Components primarily interact using the *Intent* messages. While Intent messages can explicitly address a component in an application by name, Intent versatility is more apparent for Intent messages addressed with implicit *action strings*, for which the middleware automatically resolves how to handle the event, potentially prompting the user. Recipient components assert their desire to receive Intent messages by defining *Intent filters* specifying one or more action strings.

There are four types of components used to construct applications; each type has a specific purpose. *Activity* components interface with the user via the touchscreen and keypad. Typically, each displayed screen within an application is a different Activity. Only one Activity is active at a time, and processing is suspended for all other activities, regardless of the application. *Service* components provide background processing for use when an application's Activities leave focus. Services can also export Remote Procedure Call (RPC) interfaces including support for callbacks. *Broadcast Receiver* components provide a generalized mechanism for asynchronous event notifications. Traditionally, Broadcast Receivers receive Intents implicitly addressed with action strings. Standard event action strings include "boot completed" and "SMS received." Finally, *Content Provider* components are the preferred method of sharing data between applications.

The Content Provider API implements an SQL-like interface; however, the backend implementation is left to the application developer. The API includes support to read and write data streams, e.g., if Content Provider shares files. Unlike the other component types, Content Providers are not addressed via Intents, but rather a content Uniform Resource Identifier (URI). It is the interaction between application components for which we are concerned. Figure 2 depicts common IPC between component types.

Android's application-level security framework is based on *permission labels* enforced in the middleware reference monitor [5]. A permission label is simply a unique text string that can be defined by both the OS and third party developers. Android defines many base permission labels. From an OS-centric perspective, applications are statically assigned permission labels indicating the sensitive interfaces and resources accessible at run time; the permission set cannot grow after installation. Application developers specify a list of permission labels the application requires in its package manifest; however, requested permissions are not always granted.

Permission label definitions are distributed across the framework and package manifest files. Each definition specifies "protection level." The protection level can be "normal," "dangerous," "signature," or "signature or system." Upon application installation, the protection level of requested permissions is consulted. A permission with the protection level of normal is always granted. A permission with the protection level of dangerous is always granted if the application is installed; however, the user must confirm all requested dangerous permissions together. Finally, the signature protection level influences permission granting without user input. Each application package is signed by a developer key (as is the framework package containing OS defined permission labels). A signature protected permission is only granted if the application requesting it is signed by the same developer key that signed the package defining the permission label. Many OS defined permissions use the signature protection level to ensure only applications distributed by the OS vendor are granted access. Finally, the "signature or system" protection level operates the same as the signature level, but additionally the permission is granted to applications signed by key used for the system image.

The permission label policy model is also used to protect applications from each other. Most permission label security policy is defined in an application's package manifest. As mentioned, the package manifest specifies the permission labels corresponding to the application's functional requirements. The package manifest also specifies a permission label to protect each application component (e.g., Activity, Service, etc). Put simply, *an application may initiate IPC with a component in another (or the same) application if it has been assigned the permission label specified to restrict access to the target component.*
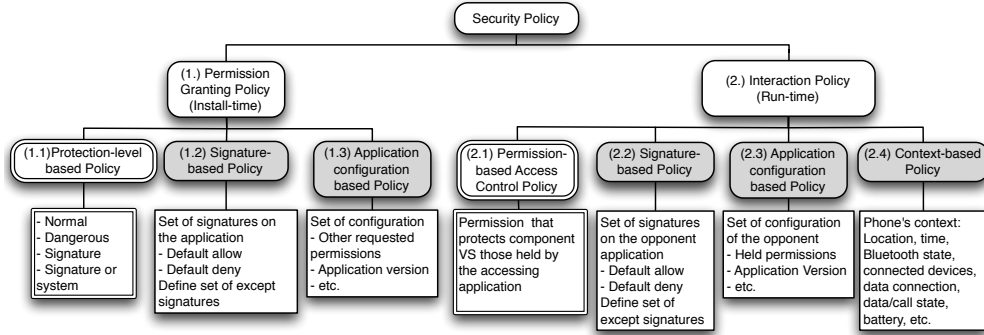
Figure 3. Policy tree illustrating the example policies required by applications. The double-stroke boxes indicate support by the existing platform.

Using this policy and permission protection levels, application developers can specify how other applications access its components. For a more complete description of the Android application level security policy and it's subtleties, see Enck et al. [6].

The permission label-based security policy stems from the nature of mobile phone development. Manually managing access control policies of hundreds (thousands) of potentially unknown applications is infeasible in many regards. Hence, Android simplifies access control policy specification by having developers define permission labels to access their interfaces. The developer does not need to know about all existing (and future) applications. Instead, the permission label allows the developer to indirectly influence security decisions. However, herein lies the limitations of Android's security framework.

IV. APPLICATION POLICIES

We explored a myriad of applications as a means of understanding the appropriate set of policy expressibility. An initial policy taxonomy is presented in Figure 3.

The permission-granting policy (1.) regulates permission assignment. In addition to controlling permission granting using Android's protection level-based policy (1.1), an application A may require *signature-based policy (1.2)* to control how the permissions it declares are granted based on the signature of the requesting application B (A and B may be signed by different developer keys). Instead, the policy grants (or denies) the permission by default with an exception list that denies (grants) the applications signed by the listed keys. An application may also require *configuration-based policy (1.3)* to control permission assignment based on the configuration parameters of the requesting application, e.g., the set of requested permissions and application version.

The interaction policy (2.) regulates runtime interaction between an application and its opponent. An application A's opponent is an application B that accesses A's resources or is the target of an action by A, depending on the access control rule (i.e., B is A's opponent for rules defined by A, and A is B's opponent for rules defined by B). Android's existing permission-based access control policy (2.1) provides straightforward static policy protec-

tion, as described in Section III. However, this policy is coarse-grained and insufficient in many circumstances. Applications may require *signature-based policy (2.2)* to restrict the set of the opponent applications based on their signatures. Similar to above, the default-allow and default-deny modes are needed. With *configuration-based policy (2.3)*, the applications can define the desirable configurations of the opponent applications; for example, the minimum version and a set of permissions that the opponent is allowed (or disallowed). Lastly, the applications may wish to regulate the interactions based on the transient state of the phone. The *phone context-based policy (2.4)* governs runtime interactions based on context such as location, time, Bluetooth connection and connected devices, call state, data state, data connection network, and battery level. Note that initially, policy types 2.2 and 2.3 may appear identical to 1.2 and 1.3; however, the former types also place requirements on the target application, which cannot be expressed with 1.2 and 1.3. However, 1.2 and 1.3 are desirable, because when applicable, they have insignificant runtime overhead.

We now present two example application policies related to our motivating example, PersonalShopper, which interacts with checkout applications, password vaults, location-based search applications, and personal ledgers.

Install-time Policy Example: In our PersonalShopper example, the location-based search application (com.abc.lbs) wants to protect against an unauthorized leak of location information from its "QueryByLocation" service. Permission granting policy can be applied when the PersonalShopper requests the permission com.abc.perm.getloc used to protect "QueryByLocation". It needs application configuration-based policy to specify that for the permission com.abc.perm.getloc to be granted, the requester must also have the "ACCESS_LOCATION" permission.

Run-time Policy Example: To ensure that the checkout application used for payment is trusted, their signatures must be checked. The PersonalShopper needs signature-based policy to specify that when the source "Personal Shopper" (com.ok.shopper) starts an Activity with action "ACTION_PAY", the policy ensures resolved applications are signed by keys in a given set.
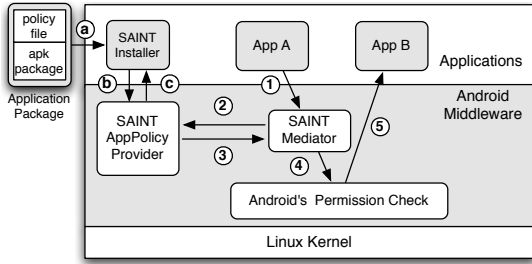
Figure 4. Saint enforcement - Saint enhances the application installation process (a-c) with additional permission granting policies and mediates component IPC (1-5) to enforce interaction policies specified by both the caller and callee applications.

V. SAINT POLICY

This section overviews the Saint policy primitives used to describe the install time policies and the interaction policies. Saint policies are those in gray boxes in Figure 3.

A. Install-Time Policy Enforcement

Saint's install-time policy regulates granting of application defined permissions. More specifically, *an application declaring permission P defines the conditions under which P is granted to other applications at install-time.* Conceptually, the an application requesting the permission P can be installed only if the policy for acquiring P is satisfied. Saint represents a substantial departure from existing Android permission assignment. The existing Android model allows/disallows a permission assignment based on application-independent rules, or where such rules provide insufficient guidance, user input. Conversely, Saint allows applications to exert control over the assignment of permissions it declares through explicit policy.

Depicted in Figure 4, install-time policies are enforced by the Saint installer based on decisions made by the AppPolicy provider, which maintains a database of all the install and run-time policies. Upon installing an application, the Saint-enhanced Android installer retrieves the requested permissions from the manifest file (step a). For each permission, it queries the AppPolicy provider (step b). The AppPolicy provider consults its policy database, and returns a decision based on matching rules (step c). If the policy conditions hold, the installation proceeds, otherwise it is aborted. Finally, on successful installation, the new application's install-time and runtime polices are appended to the AppPolicy provider's policy database.

As shown in Table I, Saint install-time policy consists of a permission label, an owner, and a set of conditions. The permission label identifies the permission to be regulated. The owner is always the application declaring the permission. The conditions are a collection of checks on the properties of the application requesting for it. All checks must be true for the installation to be allowed. The condition can check the signatures on the application package or other permissions the application requests, i.e., the permissions it would possess if installed. The condition check is implicitly affirmative in that it requires the condition to be true, e.g., as the accepted developer signatures or required set of permissions. Alternatively, it can be negated e.g., as forbidden permissions. Only the application declaring such permission is allowed to create the policy for it. The install-time policy for requirement (5) of our motivating example in Section II is provided as policy (1) in Table I. Saint encodes it in XML as:

```
<permission-grant permission="com.abc.perm.getloc"
  owner="com.abc.lbs">
  <required-permissions>
    <permission-label>
    android.permission.ACCESS_FINE_LOCATION
    </permission-label>
  </required-permissions>
</permission-grant>
```

B. Run-Time Policy Enforcement

Saint's runtime policy regulates the interaction of software components within Android's middleware framework. Any such interaction involves a *caller* application that sends the IPC and *callee* (B) application that receives that IPC. The *IPC is allowed to continue only if all policies supplied by both the caller and callee are satisfied.*

Depicted in Figure 4, the Saint policy enforcement works as follows. The caller application A initiates the IPC through the middleware framework (step 1). The IPC is intercepted by the Saint policy enforcement code before any Android permission checks. Saint queries the AppPolicy provider for policies that match the IPC (step 2). The AppPolicy provider identifies the appropriate policies, checks that the policy conditions (application state, phone configuration, etc.) satisfied, and returns the result (step 3). If the conditions are not satisfied, the IPC is blocked; otherwise, the IPC is directed to the existing Android permission check enforcement software (step 4). Android will then allow (step 5) or disallow the IPC to continue based on traditional Android policy.

Saint enforces two types of runtime policies: 1) *access* policies identify the *caller's* security requirements on the IPC, and 2) *expose* policies identify the *callee's* security requirements on the IPC. That is, access policies govern the IPC an application initiates, and expose policies govern the IPC an application receives. Note that the target (for access) and source (for expose) are implicitly interpreted as the application specifying the policy, and an application cannot specify policy for other applications.

One can view Saint policy as being similar to a network-level stateful firewall [7][3]. Like a stateful firewall, Saint identifies policy by its source and destination, and checks conditions to determine if the IPC should be allowed. In Saint, the source and destination are applications, components, Intent (event) types, or some combination thereof. Conditions are checks of the configuration or current state of the phone. Note that unlike

---

[3]A stateful firewall maintains ordered policies of the type $\{source\ address,\ destination\ address,\ flags\}$, where source and destination are IP address/ports pairs, and the flags represents the required state of the communication, e.g., whether a ongoing TCP connection between the source and destination exists.

Table I
EXAMPLE INSTALL-TIME AND RUNTIME POLICIES.

Install-time policies: (permission-label) (owner) [!]cond_1 [[!]cond_2] . . . [[!]cond_n]
(1) *(com.abc.perm.getloc) (com.abc.lbs) required-permission(ACCESS_FINE_LOCATION)*
Permission `com.abc.perm.getloc` declared by `com.abc.lbs` only be granted to applications with `ACCESS_FINE_LOCATION` permission
Run-time policies: (expose|access) (source app, type, action) (destination app, component) [!]cond_1 [[!]cond_2] . . . [[!]cond_n]
(1) *(access) (com.ok.shopper, START_ACT, ACTION_PAY) (any, any) sig:default-deny:except(3082019f3082 . . .)*
`com.ok.shopper` cannot start activity with `ACTION_PAY` action to any component in any applications unless they have signature 3082019f3082 . . .
(2) *(access) (com.ok.shopper, any, any) (com.secure.passwordvault, any) min-version(1.2)*
`com.ok.shopper` can start any interaction with any action to any component in `com.secure.passwordvault` version 1.2 or higher
(3) *(access) (com.ok.shopper, any, RECORD_EXPENSE) (any, any) forbid-permissions(INTERNET)*
`com.ok.shopper` cannot start any interaction with action "RECORD_EXPENSE" to any component in any application with permission "INTERNET"

a firewall, *all* Saint policies that match an IPC must be satisfied. Moreover, if no such policies exist, the IPC is implicitly allowed. Thus, from a technical standpoint, Saint is a "conjunctional default allow policy" rather than "default deny first match policy" [8].

As shown in Table I, Saint runtime policy consists of a type label, source application details, destination application details, and a set of conditions. The expose/access label identifies the policy type. Applications must only govern their own IPC, therefore, the specifying application must be the destination for access polices and source for expose polices. The source identifies the caller application and, if applicable, the Intent. The definition of a destination callee of a policy is somewhat more flexible.

The destination can be an application, a component, an Intent, or an application/Intent combination. We expand the notion of destination to provide finer granularity policy, as applications with many interfaces frequently require per-interface policies. For example, the security policy governing an "add an item to my shopping cart" feature provided by one component may be very different than the "authorize this transaction" component policy.

Runtime policy rules specifying multiple conditions require all conditions to be true for the IPC to proceed. Conditions can be any test that returns a Boolean value. For example, test for permission configuration, roaming state, or any other evaluation function the application deems necessary. Conditions may also be negated, indicating the IPC should only proceed when the condition is not satisfied (e.g., to blacklist configurations). For example, a reasonable policy might prevent the phone's web browser from accessing the address book or dialer. Table I provides runtime policies in response to security requirements (1), (3), and (4) of the example in Section II. In Saint, runtime policy (1) is presented in XML as:

```
<interaction direction="access">
<source>
    <application>com.ok.shopper</application>
    <interaction-type name="START_ACTIVITY" />
    <action>ACTION_PAY</action>
</source>
<destination><application>any</application></destination>
<condition>
    <signatures type="default-deny">
      <except-signature>3082019f ... </except-signature>
    </signatures>
</condition>
</interaction>
```

C. Administrative Policy

An administrative policy dictates how policy itself can be changed [9]. Saint's default administrative policy attempts to retain the best qualities of mandatory access control in Android: all application policies are fixed at installation can only change through application update (reinstallation). In Saint, the application update process removes all the of relevant policies and inserts those specified in the update. From a policy perspective, this is semantically equivalent to uninstalling and installing an application. We considered other administrative models allowing the updater to modify, add, or delete policy. However, the phone policy could unpredictably diverge from that desired by the developer quickly where, for example, update versions were skipped by the user.

There is a heated debate in smartphone operating systems community about whether to allow users to override system/application policies. A purist school of thought suggests that applications are providing MAC policies, and therefore, nothing should be changed. This provides the most stringent (and predictable) security, but potentially can prevent otherwise legitimate operations from occurring. The second school of thought says the user is always right and every policy should be overrideable.

There is no one right answer to the override debate. Hence, we introduce an infrastructure for overriding, but leave it as an OS build option. If the `SaintOverride` compile flag is set, Saint allows user override to application policy. Additionally, Saint XML policy schema includes the `Override` flag for each policy rule defined by the application. If the system `SaintOverride` system flag and `Override` flags are true, the FrameworkPolicyManager application (see Section VI) allows the user to disable the rule though the interface. If disabled, this rule is ignored during policies decisions. Note that we considered allowing the user to arbitrarily modify the policy (rather than simply disabling it), but this introduces a number of complex security and usability concerns that we defer to future work.

D. Operational Policy

Saint has the potential to hamper utility by restricting access to interfaces. Detecting such incidents is essential to be providing a useful service. Past security measures

that have prevented application behavior in an opaque and ambiguous way have not fared well, e.g., Microsoft Vista. This section defines policies that detect when Saint renders an application inefficient, faulty, or inoperable.

Consider a simple logical formulation of the Saint runtime policies. The conditions supported in the system are denoted by the set $C = \{c_1, c_2, \ldots, c_n\}$. C can be further subdivided into two sets V and T, i.e., $C = V \cup T$. V is the set of conditions which are *invariant* with respect to the system state. Invariant conditions do not change as a function of the normal operation of the phone. For example, permission assignments, developer signatures, and application version numbers are invariant. T is the set of conditions which rely on transient system state, e.g., roaming state, battery power, access to a 3G interface.

Recall that IPC is governed by the access policy p_a of the caller and the expose policy p_e of callee. A given interaction will succeed only if the conditions of both policies are satisfied. Logically speaking, each policy consists of zero or more elements of C or their negation. At any given time, the system state of the phone $\mathcal{S}$ is a truth assignment for Boolean variables for each element of C. $\hat{\mathcal{S}}$ is the set of all possible sets of $\mathcal{S}$. Let $\mathcal{V}$ be the subset of $\mathcal{S}$ relating to elements of V (the invariant conditions). The run-time IPC decision is therefore a simple test of satisfaction of the conjunction of p_a and p_e by $\mathcal{S}$, i.e., $\mathcal{S} \Rightarrow p_a \wedge p_e$.[4]

This formulation allows us to reason about the satisfiability of policy at install time. There are three possible outcomes for the install-time analysis of future IPC:

$$\mathcal{V} \Rightarrow p_a \wedge p_e \text{ (always satisfied)}$$
$$\exists \mathcal{S} \in \hat{\mathcal{S}} \mid \mathcal{S} \Rightarrow p_a \wedge p_e \text{ (satisfiable)}$$
$$\nexists \mathcal{S} \in \hat{\mathcal{S}} \mid \mathcal{S} \Rightarrow p_a \wedge p_e \text{ (unsatisfiable)}$$

where, "always satisfied" IPC will always be able to proceed (because the invariant conditions never change), "satisfiable" can occur under certain conditions (because they depend on changing system state), and "unsatisfiable" will never be allowed to occur. This last case occurs when either rule contains an unsatisfied invariant condition, e.g., incorrect developer signature, or the two rules conflict, e.g., where the expose/access rule contains a condition c and the other contains its negation $\neg c$. Note that because of the structure of the logical expressions, this satisfiability test can be tested in polynomial time (proof omitted for brevity).

We exploit this analysis to lean about the ability of an application to function. Saint tests every access rule of an application during its installation. Any rule that is unsatisfiable depicts an unusable interface, which may represent a serious functional limitation, e.g., imagine a text message application that cannot use the address book. Thus, the framework warns the user if any access rule is un-

satisfiable. Moreover, we add `FeatureRequirement` enumerated value to the XML structure of each policy rule. This value has three values; `NONE`, `AVAILABLE`, and `ALWAYS`. The `NONE` has no effect. The framework prevents the application from being installed if `AVAILABLE` is declared and the rule is unsatisfiable or if `ALWAYS` is declared and the rule is not always satisfied.

The operational policy allows the system to track and manage dependencies between applications and interfaces. By checking the operational policies of all applications during installation, update and uninstallation, we can detect when a change in an application will effect other applications. The system can warn the user or simply prevent the operation from moving forward if required interfaces become non-functional or are removed.

VI. Saint Architecture

Saint was implemented as a modification to the Android 1.5 OS. For each of the above install-time and runtime policies, we inserted one or more enforcement hooks into Android's middleware layer. In this section, we describe the relevant functionality in Android and the modifications we made to enforce Saint policies.

A. Saint Installer

The Saint installer is a modified version of Android's application installer. The installer receives the path to the downloaded Android package (.apk) and parses the package using `PackageParser` and `PackageManager`. During this step, we collect all package configurations necessary for install-time policy evaluation, such as the package's signature, requested permissions, and application version. The package's declared permissions are also acquired to verify this package's application policy.

We implement Saint's policy in a separate XML file with name identical to the package name. We chose to express the application policy in XML to match the format of Android's manifest file. Including the policy into the manifest file requires changes to the Android SDK and to the installer's package parsing function. We consider this extension as our future work.

Immediately after the package is parsed, the Saint installer examines each requested permission against its corresponding permission-granting policy queried from the AppPolicy provider. If a conflict is found, the installer rejects the installation.

After successful installation, the installer parses the application's policy file to an intermediate form. By considering the application's declared permissions obtained during the package parsing step, the installer ensures that each policy entry is inserted into the AppPolicy provider only if its permission label is declared by the application.

B. Saint Mediator

Saint's runtime enforcement covers four critical component interactions: starting new Activities, binding components to Services, receiving broadcast Intents and accessing Content Providers. For each of these interactions,

---

[4]Interfaces unprotected Saint policies are in essence "empty" policies. For the purposes of the logical analysis presented in this section, WLOG, they can be modeled simply by the Boolean value *TRUE*.

we cover the limitations of the existing Android security implementation and explain the necessary modifications and authorization hooks needed to enforce Saint policies.

Starting Activities (4.A) — As users interact with activities, they often spawn new activities for GUI elements such as menus and dialogs. In Android, a request to start a new activity takes the form of an Intent sent to the *Activity Manager Service (AMS)*, a key Android component that facilitates interactions between activities.

The AMS will then match one or more activities that have registered for that Intent. In the event that a single match is not found, i.e. there are multiple registered activities, the list of all such activities is displayed to the user who chooses the correct one, e.g. should a photograph be sent to an email client or an album. When the destination activity is known, the AMS will check if the sending activity has the permission to start such activity. If so, the activity is started. This possibility for multiple activities to match an Intent represents one of the limitations of the current Android security framework in that the registered activity has no control what component may call it beyond the permissions needed for its Intent. The calling activity has no control over which target activity is selected. To allow both the source as well as the receiver activity to influence the decision to spawn the receiver, we add a hook that restricts the set of candidate activities to choose from as shown in Figure 5.

Saint Hook Placement: If a single activity matches the Intent when it is resolved by the AMS, hook (1) checks that the conditions for both the source and destination activity before starting the destination activity as a match for the Intent. If multiple activities are registered for the Intent, it is passed to `ResolverActivity` for further Intent resolution. For each of the matched activities, hook (2) checks the source against each potential destination before allowing it to be included in the list of user options. Any destination activities not allowed by the current policy are excluded from the list. The activity selected by the user is the target activity for the Intent. There is also a small probability that only one matched activity is found. This match is checked by hook (3) whether it can be the target. Then, the target activity is started through the AMS. This time, the Intent is addressed to the specific activity and will have only a single match. The final check is performed by hook (1) to prevent TOCTTOC attack.

Receiving Intent Broadcasts (4.B) — A Broadcast Receiver acts as a mailbox for the application. It listens to Intent message broadcast by another component in the same or different application for data exchange. To specify the type of messages it is listening to, the Broadcast Receiver is attached with Intent-filter(s) that describe Intent values to be matched including the action string. Intent broadcasts are handled by the AMS, which attempts to resolve the broadcast receiver components registered for the Intent. A broadcast receiver may be registered for receiving specific Intent(s) either statically at install-time or dynamically during its execution. A static Broadcast Receiver and its permanently associated Intent-filter(s) are declared in the manifest and is always registered with the system. In contrast, a dynamic Broadcast Receiver is declared as a class in the code and is instantiated during runtime. It can be registered and unregistered any time. The Intent-filter(s) attached to the dynamic Broadcast Receiver is also created at runtime, thus can be changed.

Saint Hook Placement: In order to enforce Saint's access policies for Intent broadcasts, several authorization hooks were inserted into this process. Hook (4) is taken if the broadcast receiver is selected by name. In this case, only a single check is performed for the named receiver. If the Intent contains an action string, it can be received by potentially multiple broadcast receivers. In this case, hooks (5) and (6) iterate over the lists of potential receivers and perform a policy check for each one before it is allowed to receive a message.

Accessing Content Providers (4.C) — In Android, applications access content providers by a URI. The *Content Resolver* is responsible for mapping a URI to a specific content provider and obtaining the IPC interface to the content provider that performs the operations (e.g. query, update, etc.). Android's permission check is performed by the content provider during the operation execution. This check is inadequate to protect applications from a potentially malicious content provider that has registered under a particular URI.

Saint Hook Placement: To extend the enforcement to allow the source component to be protected as well, Saint places authorization hook (7) at the Content Resolver. The list of registered content providers is stored by the AMS in form of *Provider Record*. Therefore, our modified AMS provides the Provider Record that matches the authority string to the Content Resolver. The record contains information that allows application policy checking.

Binding Components to Services (4.D) — The last type of interaction mediated by Saint is binding a component to a service (allowing the component to access the service's APIs). A component binds to a service either by specifying its name or an Intent containing an *action string* to which that service has registered. Binding to services is mediated by the AMS, which first resolves the service by name or action string and then checks that the requesting component has the necessary permissions to bind it.

Saint Hook Placement: We inserted a single mediation point, (8), into the AMS to check Saint policy before the Android permission check. Since access policies require the source component name in the hook, we extracted the source name from a field in the binding request. For the other types of component interactions where the source name was not available, we modified the execution path up to the hook to propagate the name of the component initiating the interaction.
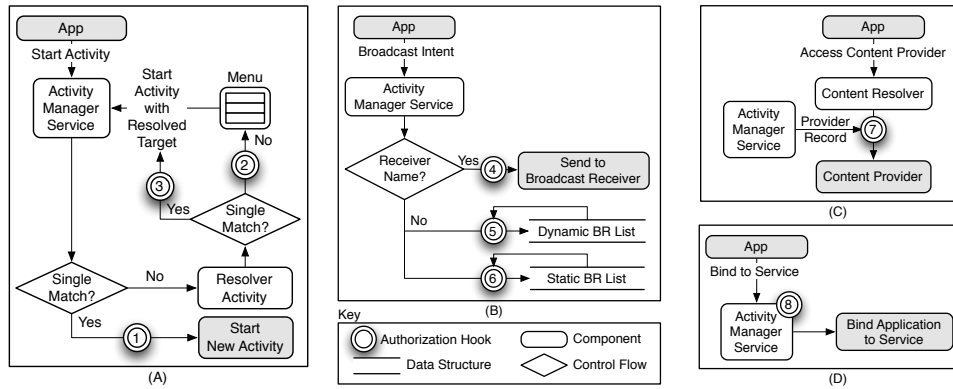
Figure 5. **Saint authorization hook placement.** The added hooks for each of the four types of component interaction are numbered (1)-(8).

C. AppPolicy Provider

The policies for both the install-time and run-time mediator are stored in the AppPolicy provider. We embedded the AppPolicy provider inside the middleware in a way similar to the phone's address book, calendar, and DRM provider are included in the platform. The policy is stored in SQLite database, which is the default database supported by Android. The database files for the provider are located in the system directory, e.g., in the `/data/system/` directory.

More importantly, the AppPolicy provider is the policy decision point. At install-time, the Saint Installer passes the information about the package being installed to the AppPolicy provider for the decision making using the exposed `verifyPermissionGrant` API. The new policy is inserted using `insertApplicationPolicy` API. Both API interfaces are implemented as part of Android's `Activity` API. At run-time, inside the middleware, the Saint mediator's hooks consult the AppPolicy provider for policy decision based on the information about the source and the destination of the interaction.

To make the decision, the AppPolicy provider retrieves all matched policies and collects all information needed to evaluate the conditions. For interaction policy, it may need to contact Package Manager and several system services such as Location Service and Telephony Service, which require the caller to run under an application environment; thus cannot be accessed by the AppPolicy provider. To address the problem, we added more functions to the AMS, which runs under "android" application environment, to obtain the information for the AppPolicy provider.

Note that it is essential to protect the API interfaces for accessing the AppPolicy provider from malicious applications. If not protected, a malicious application could simply insert bogus policies to block legitimate IPC or delete others. The current AppPolicy provider checks the identity of the application that makes the API call. If the application is not the Saint installer, the request is denied. We foresee that it may be desirable for future applications of Saint to allow other applications to view policy (e.g., policy viewers, system diagnostics). The current system will be modified to either white-list read, write, or delete

for given applications or simply check to see they have received some other system Saint policy permission.

D. FrameworkPolicyManager

As mentioned in Section V-C, FrameworkPolicyManager is implemented as an Android application to enable the user to override the policy if its `override` flag and the system's `SaintOverride` flag are true. It updates the policies in AppPolicy provider using `updateApplicationPolicy` API implemented in Android's `Activity` API. To prevent malicious applications from updating the policies, the identity of the application is checked to ensure that only the FrameworkPolicyManager can update the AppPolicy provider.

E. Condition Extensibility

So far, we have covered a set of policy enforcement mechanisms implemented by Saint. These policies are made up of conditions based on application configuration and phone state. Each condition requires code be run to inspect some aspect of either an application's context or the device's state. Currently, the AppPolicy provider is limited to the static set of implemented conditions covered in Section IV. Because we cannot predict the types of conditions future smartphone apps may wish to check in their security policies, Saint contains a generic mechanism to perform custom condition checks implemented by application developers. This mechanism works as follows.

At install time, an application's package is checked for one or more `ConditionCheckImpl` classes. These classes are instantiated and registered by Saint at boot time. The application includes conditions in its runtime policy that are handled by its `ConditionCheckImpl`. Any time a component from that application is either a source or destination of one of Saint's mediated component interactions, the condition check method of its `ConditionCheckImpl` class is called and the result is composed with the results of the Saint enforced conditions to make a policy decision. This method has the following signature: `boolean checkCondition(String condition)`, where `condition` is a custom condition string provided in the application's runtime policy and the return value is the result of the condition check.

VII. Related Work

Much of the recent work in cell phone security has centered around validating permission assignment at application installation. For example, the Kirin [10] enforces install policies that validate that the permissions requested by applications are consistent with system policy. Kirin does not consider run-time policies, and is limited to simple permission assignment. Conversely, the Application Security Framework [11] offered by the Open Mobile Terminal Platform (OMTP) recommends a certificate-based mechanism to determine the application's access rights based on its origin. Symbian offers a stricter regimen in the Symbian-signed program [12]. In this program Symbian essentially vouches for applications, and prevents unsigned applications from accessing "protected" interfaces. The MIDP 2.0 security model regulates sensitive permissions such as network access or file system access based on protection domain defined by Mobile Information Device Profile (MIDP) implementator (e.g., manufacturers and network providers) [13].

Systems for run-time policy are less developed. The Linux Security Module (LSM) framework has been frequently used to protect Linux phones. For example, the trusted mobile phone reference architecture [14] realized the Trusted Mobile Phone specification using an isolation technique developed for mobile phone platform. Muthukumaran *et al.* [15] applied SELinux security policies to Openmoko to ensure the integrity of the phone and trusted applications. In a related work, Rao *et al.* [16] developed a mandatory access control (MAC) system for smartphones which uses input from multiple stakeholders to dynamically create the policies run-time permission assignment. The Windows Mobile .NET compact framework uses security-by-contract [17] that binds each application to a behavioral profile enforced at runtime. This technique was further explored as a means for safely executing potentially malicious code [18]. Techniques such as system call interposition have also been explored for Windows Mobile [19]. None of these systems allow applications to place context-sensitive policies on both the interfaces they use and those that use their interfaces.

VIII. Conclusion

In this paper we present the Saint framework. Saint addresses the current limitations of Android security through install-time permission granting policies and run-time inter-application communication policies. We provide operational policies to expose the impact of security policy on application functionality, and to mange dependencies between interfaces. Driven by an analysis of many applications, our investigations have provided an initial taxonomy of relevant security contexts.

We are just at the beginning of this work. A most pressing need now is the integration of more applications and the policies they require into the system. We seek to extend the Saint policies to protect the phone "system" services and the cellular network, as well as integrate its interfaces with widely used security infrastructures, e.g., PKIs, enterprise systems, etc. Through ongoing feature enhancement and user study, we hope to transition Saint from a research system to a viable framework for the many millions of phones that will soon run Android.

References

[1] Apple Inc., "Apple App Store," http://www.apple.com/iphone/appstore/, June 2009.

[2] Google Inc., "Android Market," http://www.android.com/market/, June 2009.

[3] Research In Motion Ltd., "Blackberry App World," http://na.blackberry.com/eng/services/appworld/, June 2009.

[4] Independent Security Evaluators, "Exploiting android," http://securityevaluators.com/content/case-studies/android/index.jsp.

[5] J. P. Anderson, "Computer security technology planning study, volume II," Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA, Tech. Rep. ESD-TR-73-51, October 1972.

[6] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android Security," *IEEE Security & Privacy Magazine*, vol. 7, no. 1, pp. 50–57, January/February 2009.

[7] W. Cheswick, S. Bellovin, and A. Rubin, *Firewalls and Internet Security: Repelling the Wily Hacker*, Second ed. ACM Books / Addison-Wesley, 2003.

[8] P. McDaniel and A. Prakash, "Methods and Limitations of Security Policy Reconciliation," in *IEEE Symposium on Security & Privacy*, May 2002, pp. 73–87.

[9] M. Bishop, *Computer Security: Art and Science*. Reading, MA: Addison-Wesley, 2003.

[10] W. Enck, M. Ongtang, and P. McDaniel, "On Lightweight Mobile Phone Application Certification," in *Proceedings of ACM CCS*, November 2009.

[11] Open Mobile Terminal Platform (OMTP), "OMTP Application Security Framework V.2.2," pp. 1–46, 2008.

[12] Symbian Ltd., "Symbian Signed," https://www.symbiansigned.com, August 2008.

[13] Nokia Forum, "Midp 2.0: Tutorial on signed midlets v.1.1," July 2005.

[14] X. Zhang, O. Aciiçmez, and J.-P. Seifert, "A Trusted Mobile Phone Reference Architecture via Secure Kernel," in *Proceedings of the ACM Workshop on Scalable Trusted Computing*, November 2007, pp. 7–14.

[15] D. Muthukumaran, A. Sawani, J. Schiffman, B. M. Jung, and T. Jaeger, "Measuring Integrity on Mobile Phone Systems," in *Proceedings of ACM SACMAT*, June 2008.

[16] V. Rao and T. Jaeger, "Dynamic Mandatory Access Control for Multiple Stakeholders," in *Proceedings of ACM SACMAT*, June 2009.

[17] S3MS, "Security of Software and Services for Mobile Systems," http://www.s3ms.org/index.jsp.

[18] L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe, "A flexible security architecture to support third-party applications on mobile devices," in *Proceedings of ACM Workshop on Computer Security Architecture*, 2007, pp. 19–28.

[19] M. Becher and R. Hund, "Kernel-level Interception and Applications on Windows Mobile Devices," Reihe Informatik, Tech. Rep. TR-2008-003, 2008.