SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks 2012; **6**:658–673
Published online 23 August 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/sec.360

RESEARCH ARTICLE

Semantically rich application-centric security in Android

Machigar Ongtang[1], Stephen McLaughlin[2]*, William Enck[3] and Patrick McDaniel[2]

[1] Faculty of Information Technology, Dhurakijpundit University, Bangkok 10210, Thailand
[2] SIIS Laboratory, Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, U.S.A.
[3] Department of Computer Science, North Carolina State University, Raleigh, NC 17695, U.S.A.

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 are given, and areas for extension, optimization, and improvement are explored. We demonstrate through a concrete example and study of real-world applications that Saint provides necessary utility for applications to assert and control the security decisions on the platform. Copyright © 2011 John Wiley & Sons, Ltd.

KEYWORDS
security; android; smartphones

***Correspondence**
Stephen McLaughlin, SIIS Laboratory, Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, U.S.A.
E-mail: smclaugh@cse.psu.edu

1. 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, for example, Apple's App Store [5], Android's Market [14], and BlackBerry App World [25]. 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[†] operating system community and is also available in other smartphone operating systems such as Access Linux Platform [3]. Furthermore, iOS (formerly iPhone OS) has also made a step toward this application paradigm. In particular, the recently released iOS 4 supports multitasking, which allows iPhone to run multiple third-party applications simultaneously [6].

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

[†] http://www.android.com.

Copyright © 2011 John Wiley & Sons, Ltd.

Consider a hypothetical PayPal service built on Android. Applications such as browsers, email clients, software marketplaces, music players, and so on 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 which application is making the request (which in many cases in Android is unknowable to the application being called), 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 [21]:

1) *Permission assignment policy*—Applications have limited ability to control to whom permissions for accessing their interfaces are granted, for example, whitelist or blacklist 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, 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, for example, location, time, phone or network configuration, and so on. 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.

2. SMARTPHONE APPLICATION SECURITY

Figure 1 presents the fictitious PersonalShopper smartphone shopping application. PersonalShopper tracks the items a

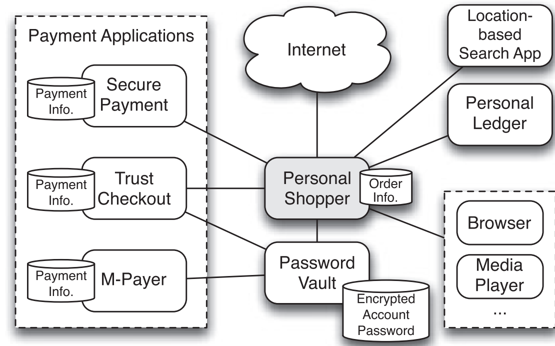

Figure 1. The PersonalShopper application finds desired items at the discretion of the user and interacts with vendors and payment applications to purchase them.

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. After 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 using unprotected WiFi connection.
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 do not have access to the Internet.
5. Applications providing functionality to PersonalShopper may also place security requirements on it. 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, for example, the phone's GPS service.

None of these policies are supported by the current Android security system. Although 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 4 examines a spectrum of policies that are potentially needed to fulfill the applications' security requirements, highlighting those that cannot be satisfied by the current Android. We then introduce goals, design, and implementation of the Saint system.

3. ANDROID

Android is a mobile phone platform developed by the Google-led Open Handset Alliance.[‡]

The platform quickly became popular among the developer community for its open source nature and adoption by telecommunications providers worldwide.

Although 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, for example, an exploited buffer overflow is restricted to the application and its data [16].

All interapplication communication passes through middleware's *binder* Inter-Process Communication (IPC) mechanism (our discussion assumes all IPC is binder IPC). Binder provides base functionality for application execution. Applications are composed of *components*. Components primarily interact using the *Intent* messages. An Intent includes two main pieces of information, namely, the address of the receiving component and the data to be passed to such component. Although Intent messages can explicitly address a component in an application by name, a more prevailing manner is to implicitly address the Intent with *action string* describing what action has occurred or what action is to be done. The action string can be one of the Android's predefined actions or a developer-defined action string. Recipient components assert their desire to receive Intent messages by defining *Intent filters* specifying one or more action strings. When the Intents are sent out, the Android middleware automatically resolves for the appropriate components for handling such event based on the declared assertion.

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 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 action strings include events for "boot completed" and "SMS received." Finally, *Content Provider* components are the preferred method of sharing data between applications. The Content Provider Application Programming Interface (API) implements an Structured Query Language (SQL)-like interface; however, the back-end implementation is left to the application developer. The API includes support to read and write data streams, for example, 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 [4]. 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

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

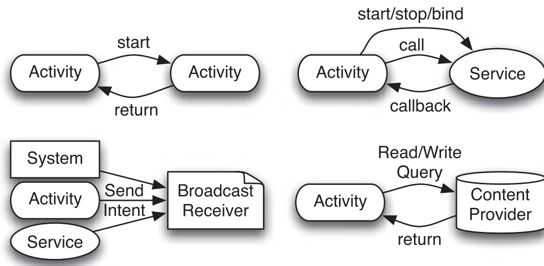

Figure 2. Typical Android application component IPC.

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 in addition, the permission is granted to applications signed by the 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*. 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 its subtleties, see the study of Enck *et al*. [13].

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, the adoption of such label-based policy has resulted in several limitations of Android's security framework as introduced in Section 1.

4. 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. Android only supports a subset of the policies classes shown in the figure, as denoted by the double-stroke boxes.

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

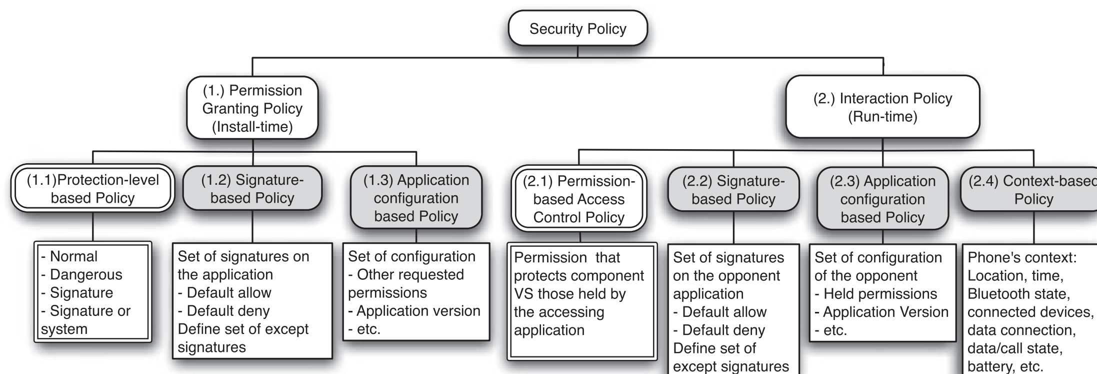

Figure 3. Policy tree illustrating the example policies required by applications. The double-stroke boxes indicate support by the existing platform.

DOI: 10.1002/sec

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, for example, the set of requested permissions and application version.

The *interaction policy* (2) regulates run-time 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 protection, as described in Section 3. 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 that mentioned, 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 run-time 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 no run-time 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.

5. 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.

5.1. 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, 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 run-time 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, that is, the permissions it would possess if installed. The condition check is implicitly affirmative in that it requires the condition to be true, for example, as the accepted developer signatures or required set of permissions. Alternatively, it can be negated, for example, 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 2 is provided as a policy in Table I. Saint encodes it in XML as follows:

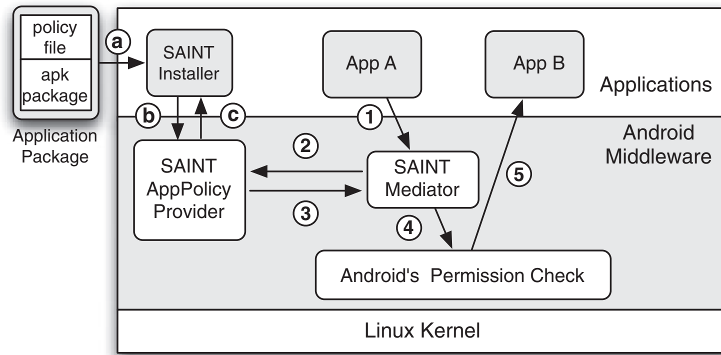

Figure 4. Saint enforcement—Saint enhances the application installation process (a-c) with additional permission-granting policies and mediates component IPC (1–5) to enforce interaction policies specified by both the caller and callee applications.

Table I. Example install-time and run-time policies.

Install-time policies: (permission-label) (owner) [!]cond1 [[!]cond2]. . .[[!]condn]
(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: (exposejaccess) (source app, type, action) (destination app, component) [!]cond1 [[!]cond2]. . .[[!]condn]
(1) (access) (com.ok.shopper, START ACT, ACTION PAY) (any, any) sig:defaultdeny: 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"

```
<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>
```

5.2. Run-time policy enforcement

Saint's run-time policy regulates the interaction of software components within Android's middleware framework. Any such interaction involves a *caller* (A) 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.) are 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 run-time policies: (i) *access* policies identify the *caller's* security requirements on the IPC, and (ii) *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 [9].[§] 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 a firewall, *all* Saint policies that match an IPC must be satisfied.

[§] A stateful firewall maintains ordered policies of the type {*source-address*,*destinationaddress*,*flags*}, where source and destination are IP address/ports pairs, and the flags represent the required state of the communication, for example, whether an ongoing TCP connection between the source and destination exists.

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" [17].

As shown in Table I, Saint run-time 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 grained 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.

Run-time policy rules specifying multiple conditions require all conditions to be true for the IPC to proceed. A condition is a test that returns a Boolean value. Condition evaluation is performed by Saint including test for permission configuration, roaming state, or other application-defined condition[¶] that 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 run-time policies in response to security requirements (1), (3), and (4) of the example in Section 2. In Saint, run-time policy (1) is presented in XML as follows:

```
<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>
```

5.3. Administrative policy

An administrative policy dictates how policy itself can be changed [8]. Saint's default administrative policy attempts to retain the best qualities of mandatory access control (MAC) in Android: all application policies that are fixed at installation can only change through application update (reinstallation). In Saint, the application update process removes of all the 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. In addition, 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 6) allows the user to disable the rule through 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.

5.4. Operational policy

Service-oriented practices usually couple security policy with operational policy keep service interactions in control [18,10]. The same concept applies to the use of Saint security 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; for example, Microsoft's User Access Control first introduced in Windows Vista has created widespread controversy [29,15]. This section defines policies that express service requirements with respect to Saint to prevent Saint from rendering an application inefficient or inoperable.

In connection with Saint, operational policies concern the usability of certain functionality requests (i.e., outgoing interactions). In our environment, the operability of each outgoing interaction is in turn governed by at most one Saint security policy included in the application and those imposed by other applications offering the required functionality. Giving such one-to-one mapping, the operability of the interaction is evaluated to true if the Saint policy rule is satisfied and the interaction is allowed.

[¶] Saint currently supports only a certain set of conditions. However, Saint distribution can be extended to support new conditions that are useful for application developers.

Security Comm. Networks 2012; **6**:658–673 © 2011 John Wiley & Sons, Ltd.
DOI: 10.1002/sec

Consider a simple logical formulation of the Saint run-time policies. The conditions supported in the system are denoted by the set $C=\{c_1,c_2,...,c_n\}$. C can be further subdivided into two sets, V and T, that is, $C=V\cup T$. V is the set of conditions that 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 that rely on transient system state, for example, roaming state, battery power, and access to a 3G interface.

Recall from Section 2 that run-time policy can take the form of the access policy p_a of the caller and the expose policy p_e of the 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 S is a truth assignment for Boolean variables for each element of C. $\hat{S}$ is the set of all possible sets of S. Let V be the subset of 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 S, that is, $S\Rightarrow p_a\wedge p_e S$.[‖]

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:

$V\Rightarrow p_a\wedge p_e$	(always satisfied)
$\exists S \in \hat{S} \vert S\Rightarrow p$	(satisfiable)
$\exists S \in \hat{S} \vert \Rightarrow p_a\wedge p_a$	(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, for example, incorrect developer signature, or the two rules conflict, for example, 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 [27].

We exploit this analysis to learn about the ability of an application to function. Saint tests every access policy of an application during its installation. Any rule that is unsatisfiable depicts an unusable interface, which may represent a serious functional limitation, for example, imagine a text message application that cannot use the address book.

Generally, operational policies can be implemented separately from security policies [18]. However, the availability of each interaction can be directly derived from the satisfiability of Saint access policy rule associated with it. Therefore, we opt to include the operational policy rule of each interaction with its corresponding Saint policy rule. More specifically, we add `FeatureRequirement` option to the XML structure of each Saint policy rule. This option specifies an operational requirement, which is one of three possible values: `NONE`, `AVAILABLE`, and `ALWAYS`. The `NONE` places no operational requirement. However, the framework warns the user if any access rule is unsatisfiable. A value of `AVAILABLE` requires that the interaction can possibly occur, namely, the Saint policy rule is satisfiable. On the other hand, a value of `ALWAYS` specifies that the interaction must always occur, that is, the rule must always be satisfied. The framework prevents the application from being installed if its operational requirements are not fulfilled.

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.

‖ 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*.

6. SAINT ARCHITECTURE

Saint was implemented as a modification to the Android 2.1 OS. For each of the mentioned install-time and run-time 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. At high level, our Saint framework consists of three main components: Saint Installer, Saint Mediator, and Framework Policy Manager.

6.1. 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.

6.2. Saint mediator

Saint's run-time 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, 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, that is, there are multiple registered activities, the list of all such activities is displayed to the user who chooses the correct one, for example, 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 TOCTTOU 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 are always registered with the system. In contrast, a dynamic Broadcast Receiver is declared as a class in the code and is instantiated during run time. It can be registered and unregistered any time. The Intent-filter(s)

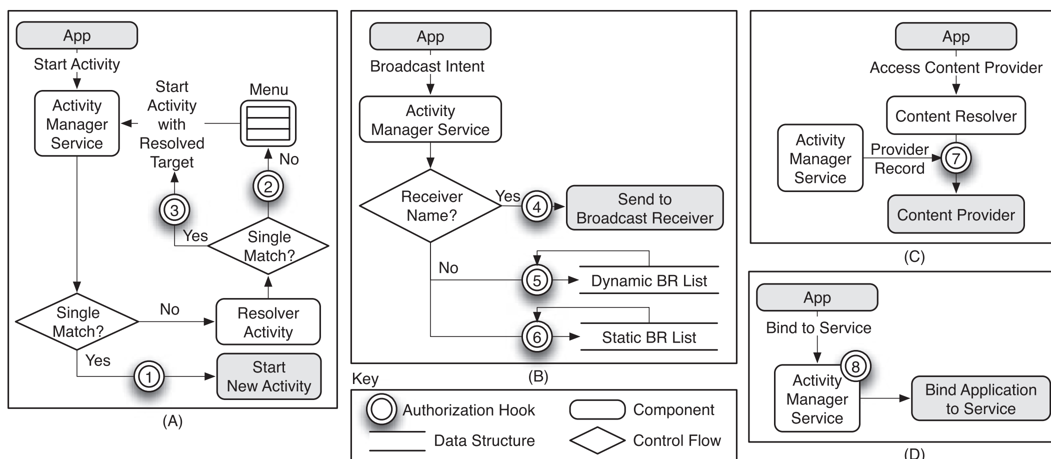

Figure 5. Saint authorization hook placement. The added hooks for each of the four types of component interaction are numbered (1)–(8).

Security Comm. Networks 2012; **6**:658–673 © 2011 John Wiley & Sons, Ltd.
DOI: 10.1002/sec

attached to the dynamic Broadcast Receiver is also created at run time and 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 potential 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 (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 the 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. Because 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.

6.3. 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 to 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 4. 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.

6.4. 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, which 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, for example, 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 user identity; thus, it cannot be accessed by the AppPolicy provider. To address the problem, we added more functions to the AMS, which runs under the "android" user identity, 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 and system diagnostics). The current system will be modified to either whitelist read, write, or delete for given applications or simply check to see they have received some other system Saint policy permission.

6.5. Framework policy manager

As mentioned in Section 3, Framework Policy Manager 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 Saint Installer and the Framework Policy Manager can update the AppPolicy provider.

7. EVALUATION

Saint provides valuable enforcement for Android application providers. We now evaluate Saint in two perspectives. First, we demonstrate Saint's value-add by discussing its applicability to applications associated with the OpenIntents collaborative project [23]. Second, we measure Saint's run-time performance overhead.

7.1. Policy appropriateness

Recall that Android applications primarily interact via Intent messages. The OpenIntents project exists to "… collect, design, and implement open intents and interfaces to make Android mobile applications work more closely together." [23]. In so doing, OpenIntents provides a registry of Intent action strings standardized by both the Android framework and participating OpenIntents application providers. Applications part of the OpenIntents project use these action strings to handle events and provide services in a consistent manner, thereby providing value-added services to one another without requiring needless reimplementation of functionality between applications. By considering the collaboration and interaction between these applications, we demonstrate the need for the Saint policy framework by existing application developers.

Table II lists sample OpenIntents Intent action strings with corresponding service caller applications, which provide the functionality and callee applications, which use the functionality (a March 2010 survey of the OpenIntents project revealed 55 unique Intent action strings and 31 caller and callee applications). For ease of exposition, we use abridged action string names (e.g., SHOW_RADAR) instead of the full namespace. In all cases shown in Table II, the Activities in the listed caller and callee applications interact to carry out the functionality. Hence, a callee is an application that handles an Intent action. For example, the Radar application contains an Activity that shows a radar graphical display upon handling the SHOW_RADAR action. Similarly, a caller is an application that makes use of functionality implemented by a callee application. For example, the Panoramio application uses the Radar

Table II. Sample OpenIntents Intent action strings with service provider and consumer applications.

Intents	Important extras	Applications
(1) com.funkyandroid.action.NEW_TRANSACTION: This intent reports a financial transaction to be recorded in an expenses or banking application	Input: Payee for a transaction Transaction amount Category of the item	Callee applications: Funky expenses
(2) com.google.android.radar.SHOW_RADAR: Display a radar like view centered around the current position and mark the given location	Input: Latitude Longitude	Callee applications: Radar Caller applications: Panoramio Photostream
(3) org.openintents.action.ENCRYPT: Data encryption functionality, which is protected by org.openintents.safe.ACCESS_INTENTS dangerous permission.	Input: URI to the file or text data to be encrypted Output: URI to encrypted file or encrypted data	Callee applications: OI Safe Caller applications: OI Notepad Obscura MyBackup Pro
(4) org.openintents.action.DECRYPT: Data decryption functionality, which is protected by org.openintents.safe.ACCESS_INTENTS dangerous permission.	Input: URI to the file or text data to be decrypted Output: URI to decrypted file or decrypted data	Callee applications: OI Safe Caller applications: OI Notepad Obscura MyBackup Pro
(5) org.openintents.action.PICK_FILE: Pick a file through a file manager.	Input: Title of text to be displayed Text on the button to be displayed Output: File URI of the selected file	Callee applications: OI File Manager Caller applications: Convert CSV Notepad
(6) android.intent.action.SEARCH: Perform a search	Input: Query string for search action	Callee applications: Collectionista WikiNotes

application to display the user's proximity to a landmark. Note that we describe these interactions from the perspective of service functionality and not the underlying mechanism, that is, the corresponding callee Activities "consume" Intent action strings. To avoid confusion, we always discuss functionality from the service functionality perspective.

The table enumerates a variety of service types: financial, location-based, cryptographic, and search functionality. In the provided examples, the Intent action string corresponds to this functionality. The listed service caller and callee applications provide use cases upon which one can discuss threat models and enforcement policy. The NEW_TRANSACTION action string handled by the Funky Expenses application provides financial ledger functionality similar to the PersonalShopper scenario. In this case, the callee application is trusted with financial data. The SHOW_RADAR action string handled by the Radar application (mentioned previously) provides location visualization for geo-tagging applications. In this case, the callee application is trusted with location information.

The ENCRYPT and DECRYPT action string handled by the OI Safe application provide cryptographic functionality. In these cases, the callee application is trusted with sensitive data, and it must trust caller applications with sensitive data it returns. For example, OI Safe is trusted not to release data encrypted by the OI Notepad application to the Obscura application. The PICK_FILE action string handled by the OI File Manager application allows the user to graphically search for a data file on SD card. In this case, the callee application is trusted to safely access the user's data and directory structure on the SD card and return the correct file selected by the user. Finally, the SEARCH action string is handled by the Collectionista and WikiNotes applications for domain specific content search. In this case, the callee applications are trusted to uphold privacy agreements made with the user.

We now demonstrate how Saint policies provide protection not expressible within the Android's existing framework. We separate corresponding policies into classes based on security goals.

Policy to Prevent Information Exposure: Caller applications frequently trust that callee applications do not expose sensitive information. Frequently, the information exposure perimeter is the phone itself. That is, the information should not leave the phone. The following policy represents how an example application, MyTrader, ensures that it uses ledger functionality only from applications that do not have Internet access. We use the notation presented in Table I.

```
"(access) (MyTrader,START_ACT,NEW_TRANSACTION) (any,any)
forbidpermission (INTERNET)"
```

Saint's information exposure policies are not limited to Internet permission checks. Saint also provides methods of whitelisting and blacklisting applications based on various features, for example, cryptographic signature and version number. For instance, any application, including the malicious ones, can offer encryption and decryption functionality. The caller application must prevent sending sensitive information to malicious applications for encryption. To prevent this, the caller application can specify that it uses only such functions offered by the applications from the developers it trusts. This requirement can be described in the Saint policy by specifying the signatures on the OI Safe application (which identifies its developer) in the set of trusted signatures. Observably, Saint provides security while preserving the flexibility of OpenIntents because the signatures on other trusted applications for encryption/decryption operations can also be included in the set. A Saint style policy representing how Obscura application can restrict the use of encryption functionality only from the applications signed by the developers that it trusts is as follows:

```
"(access) (Obscura,START_ACT,ENCRYPT) (any,any)
sig:default-deny:except (trusted-signature-set)".
```

Policy to Prevent Information Misuse: An application developer may also wish to ensure it does not inadvertently misuse information. For example, consider the SHOW_-RADAR Intent action string. The Intent takes latitude and longitude information as input. The application developer may wish to ensure that it does not misuse location information by passing it to an application that does not have permission to read location information. The following example policy allows Panoramio application to prevent such a misuse by avoiding leakage of user's locations.

```
"(access)(Panoramio,START_ACT,SHOW_RADAR)(any,any)
required-permission (ACCESS_FINE_LOCATION)".
```

Policy to Ensure Separation of Duty: Android allows third-party applications to implement and extend certain "system"-level functionality. To do so, Android assigns associated permissions with the dangerous protection level. Although users may desire such applications to be installed, application developers (and users) may desire a separation of duty when initiating communication with a callee application. The following example enables MyTrader application to ensure that it uses ledger functionality only from the applications that do not have the SET_PREFERRED_ APPLICATIONS permission, which would allow them to change the default handler without the user's knowledge.

```
"(access) (MyTrader,any,NEW_TRANSACTION) (any,any)
forbid-permission
(SET_PREFERRED_APPLICATIONS)"
```

Policy to Protect Access to Functionality: Occasionally, callee applications want to restrict access to their functionality. As an example, OI Safe application defines ACCESS_INTENT permission, which by its description allows the applications to encrypt and decrypt text, and access the passwords they have stored in OI Safe. In other words, the caller applications need to hold this permission to use such functions. The ACCESS_INTENT permission has dangerous protection level that requires user confirmation. However, it can be generously granted by the user. Allowing malicious applications to have this permission and gain

access to such functionality can introduce vulnerability. For instance, it potentially permits the malicious applications to obtain passwords set by other applications by issuing get password requests. Although access to the passwords is allowed only if the requesting applications' package names are included in the passwords' allow lists, the package names are not good authentication tokens. A malicious application can name itself as another legitimate package. It can be installed if the legitimate one is not installed on the phone. As a result, it can potentially access the password of some other applications. Saint policy can provide better protection by allowing the OI Safe application to place more restrictions on how the ACCESS_INTENT permission is granted. For instance, it can specify in the Saint policy the set of developers it trusts to use its functions as identified by the developer keys used to sign their applications. As an example, because OI Safe trusts OI Notepad, Obscura, and MyBackup Pro to use its functions, it includes the signatures on these applications in its set of trusted signature. This permission-granting policy is expressed as follows:

```
"(ACCESS_INTENT)(IOSafe)sig:default-deny:except(trusted-
signature-set)".
```

Policy to Ensure Trusted Service: Applications use utility functions offered by other applications to assist their operations. For instance, applications can use OI File Manager to select their files. Similarly, they can use search functions provided by Collectionista. In these scenarios, the applications must be able to trust that the returned and displayed results are correct. An alternative is to ensure that they actually use the functions provided by the trusted applications. Similar to the prevention of information exposure, the applications can specify the set of trusted sources of applications by identifying the developer keys used to sign them. As an example, the trusted developers of the applications for selecting file can be Google and the developers of OI File Manager and Astro File Manager, whereas for searching, they can be Google and the developers of Collectionista and OpenSearch. The signatures of the trusted developers for each action can be specified in the policy associated with it. The policies that allow an example application, MyApp, to ensure that it uses trusted select file functionality and search service can be as follows:

```
"(access)(MyApp,START_ACT,PICK_FILE)(any,any)
sig:default-deny:except
(trusted-signature-set-for-select-file)".

"(access)(MyApp,START_ACT,SEARCH)(any,any)sig:default-
deny:except
(trusted-signature-set-for-search)".
```

7.2. Performance evaluation

This section considers the implications of Saint's run-time policy enforcement for performance. We first perform macrobenchmarks to consider the human time-scale impact of Saint mediation. These show practically no human detectable slowdown between a phone running full Saint policy and one running a baseline Android installation without Saint. The overheads are then further broken down using microbenchmarks to evaluate the overhead of using SQLite to store and search Saint policies and the cost of retrieving phone state. All experiments were repeated until the 95% confidence interval was less than two orders of magnitude of the mean. It is worth mentioning for the sake of completeness that we used third-party software to increase the amount of available storage on the G1 [2] and to maintain compatibility with the Android 2.1 platform [1]. Neither of these impact Saint's performance.

Our experimental set-up involves the use of a simple inhouse benchmark that exercises each operation a minimum of 120 times. The benchmark was run with Saint configured for the developer signature test policy described in Section 5.

7.2.1. Macrobenchmarks.

In this section, we measure the affects of Saint on each of the four mediated operations: activity creation, intent broadcast, service binding, and content provider access. Table III shows the execution time of the scenarios when an activity in a caller application performs five main types of interaction with a callee application. There are 20 Saint policies of different types on the system. Both the caller and the callee application have their policies restricting the accepted signatures.

We observe that the run-time overhead of Saint is slightly under 30ms, which accounts for 5–10% of the execution time. We consider this overhead acceptable because it should not be detected by human users and only occurs during cross-components interaction. Note that for both types of starting Activity, our execution time excludes the time the screen window is composed and set visible. As a result, our reported time (for both the system with and without Saint) is slightly smaller than the time reported by the Android's Logcat tool.

7.2.2. Microbenchmarks.

For each of the interaction scenarios in Table III, our microbenchmarking reveals the underlying overhead of Saint operations as detailed in Figure 6. Most of the cost results from SQLite database operations, namely, database

Table III. Example scenarios for Saint's performance overhead.

Test case	Without Saint (ms)	With Saint (ms)	Overhead (ms) (%Overhead)
Start activity	533·69	562·83	29·14 (5·46%)
Start activity for result	477·20	506·73	29·53 (6·19%)
Broadcast intent	306·51	335·96	29·45 (9·61%)
Bind service	284·48	314·26	29·78 (10·47%)
Access content provider	293·55	323·21	29·66 (10·10%)

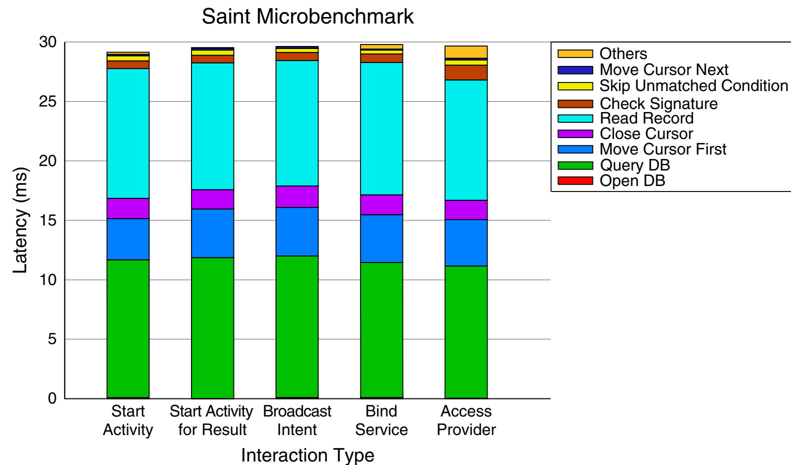

Figure 6. Microbenchmark reveals underlying cost of Saint execution.

query (11·58 ms), record reading (10·67 ms), and database cursor operations (5·741 ms).

The overhead incurred by Saint can be divided into two portions: (i) the cost from *set-up* operations, which consist of processing the interaction information, establishing and closing the AppPolicy database connection, and searching for policy statements matching the requested interaction; (ii) the cost from *policy evaluation* operations, which is composed of retrieving and processing the policies and obtaining necessary information for condition evaluation.

For each transaction, the one-time set-up cost is relatively fixed irrespective of the total number of policies in the system and the number of policies to be evaluated. For example, our experiment with the database containing up to 100 000 policies showed that the overhead from database query is relatively stable at under 12 ms. Other major one-time database operations incur approximately 5–6 ms.

In contrast to the set-up cost, the overall cost of policy evaluation increases with the number of matched policies (which are to be evaluated). Each policy evaluation involves several database operations to drop irrelevant conditions. These operations require a total execution time of approximately 5·6 ms. In addition, it also involves checking policy conditions. Such condition check overhead depends on the content of policy which reflects the type of condition information to be retrieved. Table IV shows the execution time associated with each type of condition information currently supported by Saint. Using our experimental set-up as an example, our experiment involves two matched policy records; each requires checking of signatures on the application. Therefore, each matched policy incurs about 6 ms overhead (5·6+4 ms), resulting in approximately 12-ms policy evaluation overhead.

8. RELATED WORK

Much of the recent work in cell phone security has centered around validating permission assignment at application installation. For example, the Kirin [12] enforces install policies that validate that the permissions requested by applications are consistent with the system policy. Kirin does not consider run-time policies and is limited to simple permission assignment. Conversely, the Application Security Framework [22] offered by the Open Mobile Terminal Platform 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 [28]. 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) [20].

Systems for run-time policy are less developed. The Linux Security Module framework has been frequently used to protect Linux phones. For example, the trusted mobile phone reference architecture [30] realized the Trusted Mobile Phone specification using an isolation technique developed for mobile phone platform. Muthukumaran

Table IV. Performance overhead of obtaining information for policy evaluation.

Information	Average latency (ms)
Application information	
Requested permissions	0·330
Signatures	0·4
Context information	
Data connection type	0·233
Call state	3·333
Data state	3·433
Roaming state	0·311
Location	2·967
Time	14·4

et al. [19] applied SELinux security policies to Openmoko to ensure the integrity of the phone and trusted applications. In a related work, Rao and Jaeger [24] developed a MAC system for smartphones, which uses input from multiple stakeholders to dynamically create the policies runtime permission assignment. The Windows Mobile. NET compact framework uses security-by-contract [26] that binds each application to a behavioral profile enforced at run time. This technique was further explored as a means for safely executing potentially malicious code [11]. Techniques such as system call interposition have also been explored for Windows Mobile [7]. None of these systems allow applications to place context-sensitive policies on both the interfaces they use and those that use their interfaces.

9. CONCLUSION

In this paper, we present the Saint framework. Saint addresses the current limitations of Android security through install-time permission-granting policies and runtime interapplication communication policies. We provided 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. Lastly, we demonstrated the usefulness of Saint through a case study on OpenIntents project and showed that Saint incurs only small performance overhead and thus is practical for real-world application.

A most pressing need to drive real-world adoption of Saint 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, for example, public key infrastructure and enterprise systems. 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. Android Community Rom based on Donut tree. http://www.cyanogenmod.com/, March 2010.
2. Sapphire-port-dream: Haykuro's custom ROMS for the G1. http://code.google.com/p/sapphire-port-dream/, March 2010.
3. ACCESS Co., Ltd. ACCESS Linux Platform. http://www.access-company.com/products/platforms/linux/alp.html, July 2010.
4. Anderson JP. Computer security technology planning study, volume II. Technical Report ESD-TR-73-51, Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA, October 1972.
5. Apple Inc. Apple App Store. http://www.apple.com/iphone/appstore/, June 2009.
6. Apple Inc. iOS 4. http://www.apple.com/iphone/ios4/, July 2010.
7. Becher M, Hund R. Kernel-level Interception and Applications on Windows Mobile Devices. Technical Report TR-2008-003, Reihe Informatik, 2008.
8. Becher M. *Computer Security: Art and Science*. Addison-Wesley: Reading, MA, 2003.
9. Cheswick W, Bellovin S, Ruben A. *Firewalls and Internet Security: Repelling the Wily Hacker*. Second edition, ACM Books / Addison-Wesley: Boston, MA, 2003.
10. Davies J. Oracle White Paper—Oracle SOA Suite 11g. Technical report, Oracle Corporation, January 2009.
11. Desmet L, Joosen W, Massacci F, Naliuka K, Philippaerts P, Philippaerts F, Vanoverberghe D. A flexible security architecture to support third-party applications on mobile devices. In *Proceedings of ACM Workshop on Computer Security Architecture*, 2007; 19–28.
12. Enck W, Ongtang M, McDaniel P. On lightweight mobile phone application certification. In *Proceedings of ACM CCS*, ACM: New York, NY, November 2009.
13. Enck W, Ongtang M, McDaniel P. Understanding Android security. *IEEE Security & Privacy Magazine* January/February 2009; **7**(1): 50–57.
14. Google Inc. Android Market. http://www.android.com/market/, June 2009.
15. Gruener W. Microsoft: UAC isn't broken, you just don't get it. http://www.tgdaily.com/software-features/41346-updated-microsoft-uac-i snt-broken-you-just-dont-get-it, February 2009.
16. Independent Security Evaluators. Exploiting android. http://securityevaluators.com/content/case-studies/android/index.jsp.
17. McDaniel P, Prakash A. Methods and limitations of security policy reconciliation. In *IEEE Symposium on Security & Privacy*, May 2002; 73–87.
18. Microsoft Corporation. *Application Architecture for. NET Designing Applications and Services*. Microsoft Corporation: Redmond, WA, 2002.
19. Muthukumaran D, Sawani A, Schiffman J, Jung BM, Jaeger T. Measuring Integrity on Mobile Phone Systems. In *Proceedings of ACM SACMAT*, June 2008.
20. Nokia Forum. Midp 2.0: Tutorial on signed midlets v.1.1. July 2005.
21. Ongtang M, McLaughlin S, Enck W, McDaniel P. Semantically rich application-centric security in Android. In *Proceedings of Annual Computer Security Applications Conference (ACSAC 2009)*, December 2009.

22. Open Mobile Terminal Platform (OMTP). OMTP Application Security Framework V.2.2. pages 1–46, 2008.
23. OpenIntents. Openintents. http://www.openintents.org/en/, March 2010.
24. Rao V, Jaeger T. Dynamic mandatory access control for multiple stakeholders. In *Proceedings of ACM SACMAT*, ACM: New York, NY, June 2009.
25. Research In Motion Ltd. Blackberry App World. http://na.blackberry.com/eng/services/appworld/, June 2009.
26. S3MS. Security of Software and Services for Mobile Systems. http://www.s3ms.org/index.jsp.
27. Schaefer TJ. The complexity of satisfiability problems. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, ACM: New York, NY, 1978; 216–226.
28. Symbian Ltd. Symbian Signed. https://www.symbiansigned.com, August 2008.
29. Yegulalp S. Windows Vista: Inside User Account Control (UAC). http://searchsecurity.techtarget.com.au/tips/22940-Windows-Vista-Insid e-User-Account-Control-UAC-, April 2007.
30. Zhang X, Aciiçmez O, Seifert JP. A trusted mobile phone reference architecture via secure kernel. In *Proceedings of the ACM Workshop on Scalable Trusted Computing*, ACM: New York, NY, November 2007; 7–14.