NAS-TR-0094-2008

Mitigating Android Software Misuse Before It Happens

William Enck, Machigar Ongtang, and Patrick McDaniel

22 September 2008 (updated 21 November 2008)

The Pennsylvania State University
344 IST Building
University Park, PA 16802 USA
http://nsrc.cse.psu.edu

Mitigating Android Software Misuse Before It Happens

William Enck, Machigar Ongtang, and Patrick McDaniel
Systems and Internet Infrastructure Security (SIIS) Laboratory
Department of Computer Science and Engineering
The Pennsylvania State University
{enck,ongtang,mcdaniel}@cse.psu.edu

Abstract

Mobile phones running open operating systems such as Google Android will soon be the norm in cellular networks. These systems expose previously unavailable phone and network resources to application developers. However, with increased exposure comes increased risk. Poorly or maliciously designed applications can compromise the phone and network. While Android defines a base set of permissions to protect phone resources and core applications, it does not define what a secure phone is, relying on the applications to act together securely. In this paper, we develop the Kirin security framework to enforce policy that transcends applications, called policy invariants, and provides an "at installation" self-certification process to ensure only policy compliant applications will be installed. We begin by describing the Google Android security model and formally model its existing policy. Using relatively simple policy invariants describing realistic security requirements, Kirin identified insecure policy configurations within Android leading to vulnerabilities in core phone services, thereby motivating additional security framework defining system-wide policy.

1. Introduction

Mobile phones have historically provided limited and tightly controlled interfaces to third-party applications. API restrictions often significantly limit an application's ability to interact with user data, other applications, and the network. However, this trend is reversing: open platforms such as Google Android [23], OpenMoko [41], and Apple iPhone APIs [4] provide opportunities to weave newly open phone information, communication, and location services into a wide range of novel and useful applications. This promise of innovation has inspired a surge of investment, with rapid adoption of both iPhone [27] and Android [10].

The move from closed to open mobile phone systems also raises new concerns. A single poorly vetted program can compromise user data [53], disrupt fragile cellular networks [51], [52], or simply render a mobile phone inoperable [14], [42]. The current solution offered by providers is certification [3], [34], [48]: an application may be used only if it is certified by a mandated trusted source. Online application stores such as Apple's AppStore [3] review submitted applications and certify them (or not) based in part on an evaluation of its security features and risks. Increasingly, providers [49] and OS developers [13], [37] have adopted this "AppStore" model.

Certification presents challenges in practice: the trusted third party has mixed incentives[1] that can be at odds with those of the application developer, users, and network providers. More generally, even the most altruistic certifying authority cannot fulfill its charge because (a) there is not a single definition of what is acceptable risk for all users and providers, and (b) any analysis performed without knowledge of the applications, data, and services present on a given phone will be incomplete.

The Android platform uses a modified Binder [40] framework to regulate interactions between application objects. The developers provide some initial policy in an attempt to govern system applications, but there is no way to know if the phone is secure in of itself, even less so when adding new applications. Thus, *Android does not define what a secure phone is, but rather relies on the applications to act together securely*. Therefore, additional infrastructure is required to make sense of individual application policies, interpreting them into the security requirements of the phone's many stakeholders, including the network provider, OS and application developers, and end user.

In this paper, we develop a framework, called Kirin,[2] to capture security policy that transcends Android ap-

1. There have been complaints of some providers "selectively" accepting applications based on commercial interests, rather than genuine security or quality concerns [19].

2. Kirin is the Japanese animal-god that protects the just and punishes the wicked.

plications. Stakeholders define security requirements as *policy invariants*, through which we use formal analysis to certify applications at installation; non-compliant applications will not be installed. Using relatively simple policy invariants, we have used Kirin to identify insecure policy configuration in Android's framework and bundled applications. These results lend credence to our beliefs that as mobile phone operating systems provide more open APIs for third-party applications, the security framework must not only provide per-interface permissions, but also support "administrative" policies defined by each stakeholder.

Contributions – This work makes the following contributions:

- *We reverse engineer Android's security model, providing a formal representation.* This task proved non-trivial as a result of insufficient documentation and changing security features.
- *We provide a framework for specifying and enforcing stakeholder security policy that transcends applications.* This install-time certification framework uses Prolog, a common language for security policy evaluation, to evaluate formalized application policy against policy invariants and generate automated proofs of compliance.
- *We use our framework to identify insecure application policy configurations within Android affecting voice, SMS, and location services.* We have developed proof-of-concept applications that exploit these vulnerabilities.[3]

Additionally, our experiences with Kirin have revealed important lessons for secure application development: *a*) identifying sensitive interfaces is easier than ensuring proper permission settings; *b*) applications cannot implicitly trust data in system broadcasted messages; and *c*) Kirin can help developers identify poor programming practices leading to vulnerabilities.

The remainder of this paper is structured as follows. Section 2 details the structure of the Android framework and the mechanisms used to enforce security goals. Section 3 describes an enhanced installation process. Section 4 models the Android security enforcement system within an expressive logic. Section 5 describes and demonstrates how the logical model is used to articulate security requirements through policy invariants. Section 6 evaluates Google provided applications against our invariants and demonstrates discovered weaknesses. Section 7 discusses the limitations of the model and possible extensions. Section 8 considers similar work in policy and mobile phone and network security. Section 9 concludes.

3. We are currently working with Google to use Kirin as a means of identifying flaws. This article describes our successful attempts at discovering vulnerabilities with Kirin, and our findings have lead to changes in the latest deployed release of the T-Mobile G1 firmware, and others are expected in a January release. Due to the sensitivity of outstanding security vulnerabilities, the reviewer is asked to keep the contents of this article confidential during the review process.

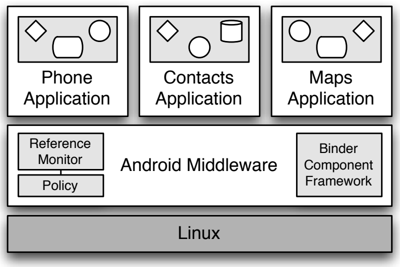


Figure 1: The Android operating system supports component-based applications using a middleware abstraction based on the Binder framework. A reference monitor mediates communication establishment between components.

2. Android Overview

The Android operating system is based upon the Linux kernel; however, it provides a middleware abstraction wherein individual applications are divided into multiple software Java "components." This section discusses the aspects of Android necessary for understanding the remainder of this paper. We assume the v1.0r1 SDK release of Android (the current release at the time of writing) unless otherwise specified.

2.1. System Architecture

The Google Android OS includes a Linux kernel, hardware drivers, application libraries, and a runtime environment based on application components that communicate through a special lightweight IPC mechanism derived from OpenBinder [40]. Applications only interact with the component framework and have no functional or security reliance on the underlying UNIX-based system (except isolation). Android defines an application as a *task* comprised of a set of components running in a pool of one or more processes, each executing one or more threads [24]. From the underlying system's perspective, each application is isolated by executing its processes as a unique UNIX user identity (*uid*); however, the runtime environment enables application interaction (e.g., sharing objects, executing methods, registering callbacks, etc.) through the Binder mechanism, which ensures data object consistency across processes, similar to COM [36] and CORBA [50]. The Android middleware relays messages and resolves application names to establish *Inter-Component Communication* (ICC), Android's dual to

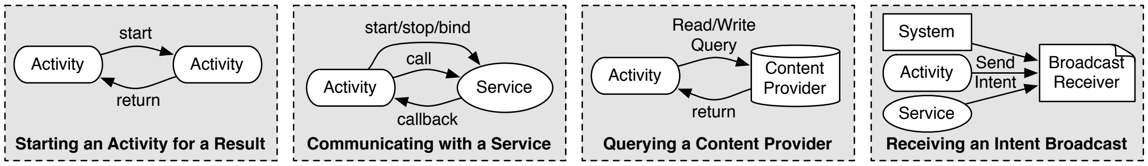


Figure 2: Typical ICC between Activities, Services, Content Providers, and Broadcast Receivers.

IPC on traditional UNIX-like systems. Shown in Figure 1, a reference monitor [2] mediates ICC establishment according to a system policy; however, once ICC is established, communication proceeds uninhibited.

An application consists of multiple components defined in a manifest file; there is a 1-to-many relationship between applications and components. Shown in Figure 2, there are four component types,[4] and an application may contain any number or combination. An *Activity* component interfaces with the physical user via the touchscreen and keypad. By convention, an application will contain many Activities, one for each "screen" presented to the user. The interface progression is a sequence of one Activity "starting" another, possibly expecting a return value. A *Service* component provides background processing that continues even after its application loses focus. Services also establish arbitrary interfaces for ICC, including method execution and callbacks, which can only occur after the service has been "bound". A *Content Provider* component is a database-like mechanism for sharing data with other applications within the system. This interface supports standard SQL-like queries, e.g., `SELECT`, `UPDATE`, `INSERT`, through which components in other applications can retrieve and store data according to the Content Provider's schema. A *Broadcast Receiver* component acts as an asynchronous mailbox for directed broadcasts of system and application event messages. Once a Broadcast Receiver is registered with the system to receive a certain type of message, it will be executed by the system as needed. The message itself arrives in the form of an *Intent*, which is Android's data primitive for most ICC initiation. An Intent simply bundles data with a target component address, defined either explicitly by component name or implicitly by an "action" string, which is used by the system to resolve the appropriate destination component, launching the application if necessary.

2.2. Security Enforcement

In its most basic operation, shown in Figure 3, Android's middleware reference monitor provides manda-

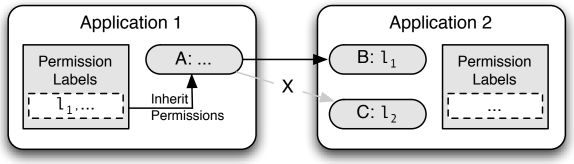


Figure 3: Android's basic access control model defines how applications can access components. Application 1 is assigned label l_1 (but not l_2), which is inherited by Component A. Access to Components B and C is restricted by permission labels l_1 and l_2, respectively. Therefore, Component A can access B but not C.

tory access control (MAC) enforcement of how applications access components, with the assumption that all inter-application communication occurs through the described application-level ICC (the underlying UNIX system provides file and IPC isolation). The reference monitor decides access based on a *permission label*[5] primitive. An application's manifest definition enumerates the set of permission labels it uses at runtime (these permission labels are granted at install-time and cannot change afterwards). Conversely, the manifest definition of a component specifies a single access permission label used to restrict runtime access (two components may share the same access permission label). At runtime, the component initiating ICC inherits all permission labels assigned to its containing application, and if the label restricting access to the target component is in that set, ICC may proceed, otherwise access is denied.

Unfortunately, Android has extended its conceptually simple MAC enforcement model with a number of complex exceptions and extensions that must be understood by application developers. The security extensions operate as follows:

Public vs. Private Components – Similar to Java class methods, not all components are addressable from an external application. For example, an application may contain many "sub-Activities" that should only be started from other components within that application (e.g., they return sensitive data). A component is public or private as a result of either a) an explicit "exported" tag in the manifest, or b) implicit rules

4. We capitalize the first letter of component types to distinguish them from terms describing general concepts.

5. The Android documentation refers to permission labels as simply permissions; however, we include the term "label" to distinguish the text string from the semantics of having a permission.

interpreted by Android.

Implicitly Open Components – The manifest definition of a public component need not specify an access permission label (by default no label is specified), thereby allowing access to any application on the system. This configuration is required for the "main" Activity to allow the application to be "launched."

Protected APIs – The Android SDK includes special "reserved" permission labels for accessing hardware and other resources. The most notable permission protects the network interface. An application must be assigned the `INTERNET` permission label for its components to make network connections.

Content Provider Permissions – The manifest definition of a Content Provider allows the developer to define both read and write access permission labels to restrict queries that read (e.g., `SELECT`) and write (e.g., `INSERT`), respectively.

Intent Broadcast Permissions – When an application broadcasts an Intent, the API optionally allows the sender to include an access permission label. To receive such an Intent, the application containing the Broadcast Receiver must include that permission label. This effectively limits the applications that can receive the broadcasted message.

Service Hooks – Once a Service component is bound to, the initiating component can execute any method defined in that Service's interface. The developer may extend an application with additional reference monitor hooks in the form of `checkPermission()` method invocations. These hooks allow the developer differentiate access to interface methods.

Permission Protection Levels – The Android framework provides a base set of permission labels; however, application developers may define additional permissions. Each permission definition includes "protection level" meta-information used when granting an application the permissions it requests. There are four protection levels: "normal," "dangerous," "signature," and "signature or system". "Normal" permissions are always granted; "dangerous" permissions are granted only after user confirmation; "signature" permissions are granted only if the requesting application is signed with the same developer key as the application defining the permission; and "signature or system" permissions follow the rules of "signature" permissions, with the exception of always being granted to applications installed in `/system/app`.

Pending Intents – An application can define an Intent with the intention of performing ICC, but instead create a "Pending Intent" object. By passing this object to another application, the originating application delegates the ability to fills in remaining fields (e.g., destination, or data) and time of ICC execution. Upon executing the Pending Intent, an RPC invokes the ICC from the originating application.

URI Permissions – Content Providers are addressed by a URI of the form content://authority/table/[id], where "authority" specifies the Content Provider, "table" indicates a table within the database, and "id" optionally specifies a record. An application with read or write access to a Content Provider can specify a URI in the "data" field of an Intent with additional read or write "grant" flags set, thereby delegating the ability to read or write that record in the Content Provider to the recipient of the Intent, even though the recipient may not have read or write permission for the Content Provider.

2.3. Example

Figure 4 illustrates a concrete policy evaluation. The Android SDK includes the *ContactsProvider* application that contains one component, a Content Provider similarly named ContactsProvider, which is used by all applications to lookup phone numbers and store contact information for the user's acquaintances. The ContactsProvider component is assigned the read permission label `READ_CONTACTS` and the write label `WRITE_CONTACTS` in order to restrict read and write requests, respectively. The SDK also includes a *Contacts* application that contains a number of Activities that allow the user to manipulate the address book. The Contacts application is assigned both the `READ_CONTACTS` and `WRITE_CONTACTS` permission labels in its manifest file. Therefore, any Activity component in the Contacts application can read and write entries in the ContactsProvider component. The SDK also includes the *Maps* application, which is only assigned the `READ_CONTACTS` permission label. Activity components within the Maps application can query the ContactsProvider to find an address; however, they cannot write back to the address book. Here, the policy attempts to embody *least privilege* [45].

3. Kirin

The complexity underlying Android's security model makes it impossible for even sophisticated users to reason about overall phone security. We propose a model wherein before the system installs a downloaded application package, it first ensures the application meets predefined security requirements. If any requirements are not met, the package is rejected. This installation model ensures the mobile phone stays in its original secure state, without relying on the user to make security decisions.

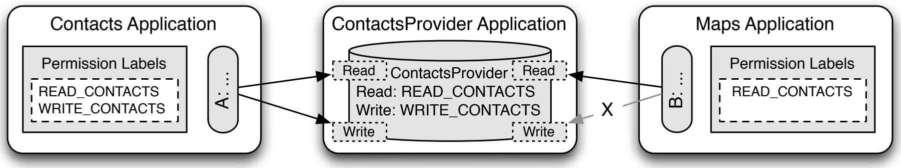


Figure 4: Policy evaluation for the Contacts example.

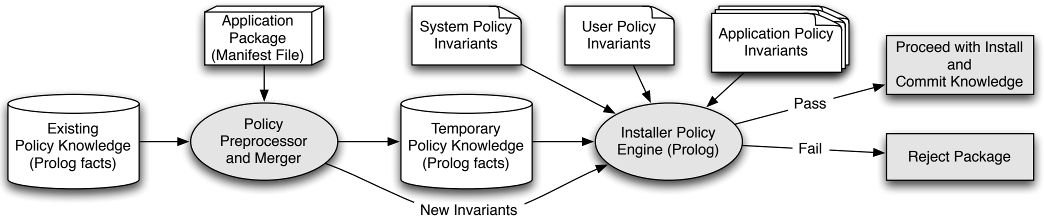

Application Package (Manifest File)
System Policy Invariants
User Policy Invariants
Application Policy Invariants
Proceed with Install and Commit Knowledge
Existing Policy Knowledge (Prolog facts)
Policy Preprocessor and Merger
Temporary Policy Knowledge (Prolog facts)
Installer Policy Engine (Prolog)
Pass
Fail
Reject Package
New Invariants

Figure 5: Enhanced Installation Logic. New packages cannot be installed unless all policy invariants pass.

Figure 5 illustrates Kirin's enhanced installation process. A policy preprocessor extracts policy (e.g., the permission assignments described in Section 2.2) from the target application package, turning it into Prolog facts, and merges it with existing policy knowledge. The result of the merger represents the system's security state if the package installation were to proceed. The policy engine then uses the temporary policy state to evaluate invariants defined by the key stakeholders. The policy invariants are encoded as Prolog predicates that perform queries on the temporary policy state composed of Prolog facts. In doing so, we use Prolog to automatically generate compliance proofs for the target application. If all invariants pass, the installation proceeds, and the permanent policy store is appended; however, if any invariant fails, the package is rejected.

The remainder of this paper describes the theory and constructions required to build Kirin. We begin with a formal interpretation of Android's security model.

4. Formal Interpretation

Kirin's ability to reason about policy requires formal semantics to be extracted from Android's security model. We begin by modeling Android's basic MAC enforcement model and then capture the above mentioned extensions through a series of preprocessing steps the retain the expressibility of the basic model. Note, we only model ICC enforcement, as Android provides low-level isolation. Additionally, we defer placing restrictions on the first-order logic used to express the model until Section 4.3.

4.1. Policy Model

We initially express an Android security policy as a traditional Subject-Object-Rights (SOR) Access Matrix [30]. Each application includes a manifest file that defines the contained components, permissions required to access components, permissions requested by the application, and potentially newly defined permissions. The aggregate of system and add-on application manifests defines the system policy. We model this policy as $P : S \times O \times R \rightarrow \{\texttt{true}, \texttt{false}\}$, where `true` and `false` represent allow and deny, respectively.

Subjects (S) – The subjects are the set of all *applications* in the system. In Android, permissions are assigned to an application and inherited by all contained components. All access control decisions are made with respect to the application containing the component initiating the ICC.

Objects (O) – The objects are the set of all *components* in the system. In Android, access permissions are assigned to components, hence they are the objects in the access matrix.

Rights (R) – The rights are the set of all *permission labels* in the system. Figure 6 depicts rights assignment. There is a 1-to-many mapping between subjects and rights; mapping an application to a permission label assigns it that right. There is a many-to-1 mapping between objects and rights; mapping a component to a permission label requires the subject to have that right to access the component.

Policy $P(\cdot)$ – Trivially, the policy allows a component in application $s \in S$ to access a component $o \in O$ that requires right $r \in R$ if the following expression evaluates to true:

$$P(s,o,r) = \texttt{requires}(o,r) \wedge \texttt{has\_perm}(s,r)$$

where $\texttt{requires}(o,r) = \texttt{true}$ when the manifest file for component o specifies r as the access permission, and $\texttt{has\_perm}(s,r) = \texttt{true}$ when application s is assigned r in its manifest file. The intuition to use permission labels as rights stems from both

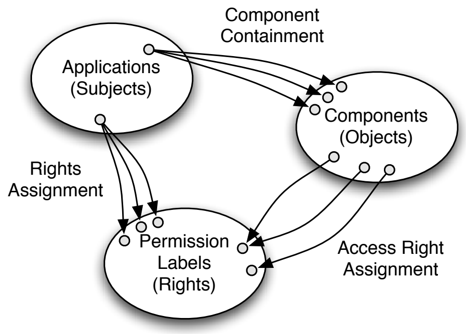


Figure 6: Subjects, Objects, and Rights Relationships

the natural language description of the labels, e.g., WRITE_CONTACTS, and the semantics of assigning "rights" to an application that are required to access specific components throughout the system. Finally, we say $\texttt{contains}(s, o) = \texttt{true}$ when application s contains component o.

4.2. Capturing Model Extensions

We now turn to the security model extensions described in Section 2.2. We capture these extensions by expanding the sets of objects and rights through a series preprocessing steps.

Public vs. Private Components – As private components are not addressable by external applications, we mimic Android's inference rules and exclude private components from the model.

Implicitly Open Components – We model components with unspecified access permission labels by assigning a special reserved "open" right, ε, as follows:

1) If the manifest has not specified an access permission for a component, assign ε to the corresponding object o, i.e., $\texttt{requires}(o, \varepsilon) = \texttt{true}$.
2) Assign ε to all subjects corresponding to applications, i.e., $\forall s \in S, \texttt{has\_perm}(s, \varepsilon) = \texttt{true}$.

By inspection, this modification gives all applications access to components without explicitly specified access permissions while retaining the monotonicity of our policy model.

Protected APIs – We model each protected API by adding one object o_i and one right r_i and ensuring $\texttt{requires}(o_i, r_i) = \texttt{true}$, for example, $\texttt{requires}(\texttt{network}, \texttt{INTERNET}) = \texttt{true}$. Note that no application contains these special objects.

Content Providers Permissions – For each Content Provider o, replace o with two new objects o_r and o_w corresponding to the read and write interfaces, respectively. Then, ensure $\texttt{requires}(o_r, r_r) = \texttt{true}$ and $\texttt{requires}(o_w, r_w) = \texttt{true}$ for the read (r_r) and write (r_w) permission labels specified in the manifest.

Intent Broadcast Permissions – Recall that Intent broadcasts can address either an explicit destination (i.e., a single Broadcast Receiver), or more commonly, an implicit destination (via an "action" string to which any number of Broadcast Receivers can "subscribe"). To capture read permissions on explicit destinations, we create a new object $o_{b,r}$ for each Broadcast Receiver b and right r. To capture permissions on implicit destinations, we create a new object $o_{a,r}$ for each action string a and right r. We then ensure $\texttt{requires}(o_{b,r}, r) = \texttt{true}$ and $\texttt{requires}(o_{a,r}, r) = \texttt{true}$ for each new object. This object expansion ensures the model captures every possible access control case (i.e., maximal access). However, for practical purposes, only Intent objects of interest are included in the policy knowledge; Section 7.1 discusses the possibility of automatic enumeration via code analysis.

Service Hooks – While hook placement can imply arbitrary access policies, we believe common practice will assign one access permission per Service interface method. Assuming this operation, we model this extension by creating a new object o_i for each interface and ensure $\texttt{requires}(o_i, r) = \texttt{true}$ as appropriate. However, like broadcast permissions, code analysis is required to extract policy semantics, therefore our current implementation conservatively ignores per-interface service hooks to view Services as opaque objects at the cost of false positives. Section 7.1 discusses how simple manifest extensions can resolve this complication.

Permission Protection Levels – No changes are required to model "normal" permissions. We model "dangerous" permissions by ensuring the $\texttt{has\_perm}(s, r)$ fact is removed if the user denies the assignment (Section 6 assumes "Click OK to proceed." succeeds). "Signature" and "signature or system" permissions are modeled by expanding the set of rights and modifying assignment as follows.

For each "signature" right $r \in R$:

1) Create the right r_{ki} as "sig_<hash(ki)>:r" for each signature key k_i, where <hash(ki)> is the hash (e.g., SHA1) of k_i (public key).
2) $\forall s \in S$, if $\texttt{has\_perm}(s, r) = \texttt{true}$, ensure $\texttt{has\_perm}(s, r_{ks}) = \texttt{true}$, where k_s is s's signature key.
3) $\forall o \in O$, if $\texttt{requires}(o, r) = \texttt{true}$, set it to $\texttt{false}$ and ensure $\texttt{requires}(o, r_{ki}) = \texttt{true}$, where k_i is the key used to sign the application defining r.

This preprocessing ensures s can only access o if $k_s = k_i$ and $P(s, o, r)$ was previously $\texttt{true}$.

For each "signature or system" right $r \in R$:

1) Create the right r_s as "`system:r`".
2) $\forall s \in S$, if $\texttt{has\_perm}(s, r) = \texttt{true}$ and s is in `/system/apps`, ensure $\texttt{has\_perm}(s, r_s) = \texttt{true}$.
3) $\forall o \in O$, if $\texttt{requires}(o, r) = \texttt{true}$, ensure $\texttt{requires}(o, r_s) = \texttt{true}$.
4) Ensure all of the rules defined for "signature" permissions hold.

This preprocessing ensures system applications can always access objects protected by "signature or system" permissions. Finally, if s is installed in `/system/apps`, we create the reserved right `system` and ensure $\texttt{has\_perm}(s, \texttt{system}) = \texttt{true}$ for use in policy invariants.

Pending Intents and URI Permissions – Both Pending Intents and URI permissions are late additions to Android's security framework, first appearing in the v0.9r1 (August 2008) and v1.0r1 (September 2008) SDK releases. Incorporating permission delegation has far reaching implications for any access control model, especially so when done as an afterthought. Future work will investigate the impact of these mechanisms.

4.3. Invariant Building Blocks

We now demonstrate how our model can express higher level security requirements. We construct invariants as non-recursive queries over the knowledgebase of `has_perm`, `requires`, and `contains` facts using first-order logic with negation as failure (NAF) and the closed world assumption. This constraint is a direct result of the types of security requirements discussed in Section 5. The remainder of this section describes three different *invariant patterns* that sample types of policy expressibility one might expect from mobile phone security requirements, but are not meant to be all encompassing. While these examples are in fact invariants, we term them patterns for differentiation from the invariants presented in Section 5. For demonstration purposes, we use the example in Section 2.3 with the additional assumption that the Maps application has Internet access. See the appendix for a Prolog encoding of these patterns. Each pattern is defined as $\texttt{pat}_X : S \rightarrow \{\texttt{true}, \texttt{false}\}$.

4.3.1. Simple Access Control Checks. The most trivial pattern is a formula describing if a subject can access an object. For example, "Can application s acquire write access to ContactsProvider?". $\texttt{pat}_1$ encodes this question in logic.

$$\texttt{pat}_1(s) = \exists r \in R.\ P(s, \texttt{ContactsProvider\_w}, r)$$

Informally, this formula asks if application s can use any right r to access the write interface of `ContactsProvider`. In our Contacts example, the Contacts application satisfies $\texttt{pat}_1$, because it has the `WRITE_CONTACTS` permission; however, the Maps application does not, and hence does not satisfy $\texttt{pat}_1$.

4.3.2. Mutual Exclusion. Security requirements commonly embody Chinese Wall [11] characteristics wherein an application cannot have simultaneous access to resources. For example, "Can application s be installed such that if it can read from ContactsProvider, then it cannot connect to the network?". $\texttt{pat}_2$ encodes this question in logic.

$$\begin{aligned}\texttt{pat}_2(s) = &\forall r_1 \in R, \exists r_2 \in R, \forall r_3 \in R.\\ &\quad \neg P(s, \texttt{ContactsProvider\_r}, r_1)\\ &\quad \vee (P(s, \texttt{ContactsProvider\_r}, r_2)\\ &\qquad \wedge \neg P(s, \texttt{network}, r_3))\end{aligned}$$

Informally, this formula ensures that either application s cannot read from `ContactsProvider` using any right r_1, or if there exists some right r_2 allowing s access, then s cannot access the network with any right. In our Contacts example, the Contacts application satisfies $\texttt{pat}_2$, because it does not have Internet access; however, the Maps application has Internet access, therefore it does not satisfy $\texttt{pat}_2$.

4.3.3. Rights Dependence. A variation of mutual exclusion is a dependence condition, a useful requirement for ensuring proper operating conditions (i.e., usability). For example, "Can application s be installed such that if it can write to ContactsProvider, then it can also read from it?" $\texttt{pat}_3$ encodes this question in logic.

$$\begin{aligned}\texttt{pat}_3(s) = &\forall r_1 \in R, \exists \{r_2, r_3\} \in R.\\ &\quad \neg P(s, \texttt{ContactsProvider\_w}, r_1)\\ &\quad \vee (P(s, \texttt{ContactsProvider\_w}, r_2)\\ &\qquad \wedge P(s, \texttt{ContactsProvider\_r}, r_3))\end{aligned}$$

Informally, this formula ensures that either application s cannot write to `ContactsProvider` using any right r_1, or there exists a pair of rights r_2 and r_3 that allow both write and read access. In our example, both `ContactsApp` and `MapsApp` satisfy $\texttt{pat}_3$.

5. Realistic Policy Invariants

We now use the formalism and invariant building blocks described in Section 4 to define policy invariants representing realistic security requirements for mobile phones. For illustrative purposes, we describe invariants protecting core system functionality (e.g., making phone calls), user privacy (e.g., access to call information), and applications (e.g., preventing forged messages). The invariants are not necessarily compatible and were designed to demonstrate desirable concepts (e.g., privacy) currently unavailable in Android. When applied correctly, Kirin can use these invariants to identify applications asserting harmful sets of permissions or discover insecure policy configuration within applications or the platform itself.

The invariants described herein are not intended to exhaustively define appropriate security requirements for mobile phones; this can only be done by the stakeholders themselves. Similarly, the invariants are not intended to test the limits of our model's expressibility. Rather, we have defined relatively simple, but practical, invariants to show the power and usefulness of an automated analysis framework such as Kirin.

Similar to the patterns in the previous section, invariants take the form $\mathtt{inv}_X : S \rightarrow \{\mathtt{true}, \mathtt{false}\}$.

5.1. Core System Functionality Invariants

Invariant 1: *"An application must have explicit permission to make an outgoing voice call."*

Description: While the Android framework uses the `CALL_PHONE` and `CALL_PRIVILEGED` permissions to protect the API for making outgoing calls, an application given a call permission may indirectly provide API access via a component interface (e.g., starting an Activity). This Invariant ensures applications do not inadvertently allow indirect access.

Logic: The system developer (e.g., Google) knows all interfaces for making voice calls. For each such interface object $\mathtt{IF}_i$, including both the telephony API and all relevant components in the system provided `PhoneApp`, we define $\mathtt{inv}_{1i}(s)$ that must be true for all applications $s \in S$ installed on the system.

$$\begin{aligned}\mathtt{inv}_{1i}(s) = & \forall r_1 \in R, \exists r_2 \in R.\ \neg P(s, \mathtt{IF}_i, r_1) \vee \\ & (P(s, \mathtt{IF}_i, r_2) \wedge (\mathtt{has\_perm}(s, \mathtt{CALL\_PHONE}) \\ & \vee\, \mathtt{has\_perm}(s, \mathtt{CALL\_PRIVILEGED})))\end{aligned}$$

Informally, this formula states that if an application s can access the interface $\mathtt{IF}_i$ capable of directly making phone calls from input data, then it must have one of the special call privileges. A violation of this invariant indicates an interface on the phone that could be exploited by an application without proper permissions. Pragmatically, such a policy not only restricts direct access to the call setup network interfaces, but also to "dialer" applications.

Invariant 2: *"An application holding a dangerous permission must have no unprotected components."*

Description: The Android framework tags a subset of the provided permissions with the "dangerous" protection level. Semantically, this meta-information requires user confirmation for any application requesting it. For example, the `CALL_PHONE` and `RECEIVE_SMS` permissions are marked as dangerous to ensure the user is notified when a new application wishes to make outgoing phone calls and receive SMS messages. This invariant prohibits a new application from creating an open interface through which other applications can indirectly gain access to "dangerous" functionality.

Logic: For each "dangerous" permission label $\mathtt{PL}_i$ in the framework (and accompanying applications), we define $\mathtt{inv}_{2i}(s)$ that must be true for a new application s to be installed.

$$\begin{aligned}\mathtt{inv}_{2i}(s) = & \forall o \in O.\ \neg\mathtt{has\_perm}(s, \mathtt{PL}_i) \vee \\ & (\mathtt{has\_perm}(s, \mathtt{PL}_i) \wedge \\ & \neg(\mathtt{contains}(s, o) \wedge \mathtt{requires}(o, \varepsilon)))\end{aligned}$$

Informally, this formula states that an application s with the permission label $\mathtt{PL}_i$ must not contain an implicitly open component.

Invariant 3: *"Only system applications can interface with hardware."*

Description: The Android framework provides high level Java APIs for interfacing with hardware. To provide flexibility, Android allows any application with the proper permissions to use these APIs. Commonly, only system applications use these APIs, providing add-on applications indirect access through exported service interfaces. This invariant ensures that only system applications obtain direct access.

Logic: The framework developer knows all hardware interfaces. For each such interface object $\mathtt{IF}_i$, we define $\mathtt{inv}_{3i}(s)$ that must be true for an application s to be installed on the system.

$$\begin{aligned}\mathtt{inv}_{3i}(s) = & \forall r_1 \in R, \exists r_2 \in R.\ \neg P(s, \mathtt{IF}_i, r_1) \vee \\ & (P(s, \mathtt{IF}_i, r_2) \wedge \mathtt{has\_perm}(s, \mathtt{system}))\end{aligned}$$

Informally, this formula states that if an application s can access a hardware interface $\mathtt{IF}_i$, then it must have preprocessed the `system` right.

5.2. User Privacy Invariants

Invariant 4: *"Only system applications can process outgoing calls."*

Description: The Android framework allows applications to receive notification of outgoing calls, including the destination phone number. For privacy reasons, a user may wish to specify that only system applications (i.e., in `/system/apps`) may receive such notifications. The notification itself is an Intent broadcasted to the `NEW_OUTGOING_CALL` action string restricted by the `PROCESS_OUTGOING_CALLS` permission.

Logic: This invariant relies on two policy extensions described in Section 4.2. First, we model the Intent notification message as the combination of the action string ($\mathtt{NEW\_OUTGOING\_CALL}$) and the permission label (PROCESS_OUTGOING_CALLS). We refer to this object as $\mathtt{I}_{out}$ for simplicity. Second, we use our preprocessing rule indicating that an application has the system right if it is installed in /system/apps. These two extensions allow us to define $\mathtt{inv}_4(s)$ that must true to install applications s.

$$\mathtt{inv}_4(s) = \forall r_1 \in R, \exists r_2 \in R.\ \neg P(s, \mathtt{I}_{out}, r_1) \vee (P(s, \mathtt{I}_{out}, r_2) \wedge \mathtt{has\_perm}(s, \mathtt{system}))$$

Informally, this formula states that if an application s can access the $\mathtt{I}_{out}$ Intent object, it must have the preprocessed system right.

Invariant 5: *"Applications that can perform audio record must not have network access or pass data to an application that has network access."*

Description: Privacy conscious users may desire separation between the audio recording subsystem and the Internet. While modeling information flow policies in Android is difficult (see Section 7.2), we can define an invariant that mitigates Internet-based eavesdropping by looking at an application and all connecting applications (three colluding applications may be unlikely).

Logic: As discussed in Section 4.2, resource APIs are modeled as additional objects; this invariant uses the record_audio object. We define $\mathtt{inv}_5(s)$ such that it must be true to install application s.

$$\begin{aligned}\mathtt{inv}_5(s) = &\forall r_1 \in R, \exists r_2 \in R, \forall r_3 \in R, \forall s_1 \in S, \forall o \in O,\\ &\forall \{r_4, r_5\} \in R.\ \neg P(s, \mathtt{record\_audio}, r_1)\\ &\vee (P(s, \mathtt{record\_audio}, r_2) \wedge\\ &\quad \neg P(s, \mathtt{network}, r_3) \wedge\\ &\quad \neg (P(s, o, r_4) \wedge \mathtt{contains}(s_1, o) \wedge\\ &\qquad P(s_1, \mathtt{network}, r_5))\ )\end{aligned}$$

Informally, this formula states that if an application s can access the audio record API, then it cannot access the network or an application with network access.

Invariant 6: *"An application with access to wifi or network state must also declare network access."*

Description: The Android framework provides an API (protected by the ACCESS_NETWORK_STATE permission label) that registers a callback that executes on network state change (e.g., GPRS vs HSPA, ESSID). Network state, especially an ESSID, leaks information about the user's physical activity. This invariant ensures an application with network state access also has access to the network. A similar invariant can be written for access to Bluetooth and Bluetooth state.

Logic: This invariant models the network state API the network_state object. We define $\mathtt{inv}_6(s)$ that must be true for an application s to be installed.

$$\begin{aligned}\mathtt{inv}_6(s) = &\forall r_1 \in R, \exists \{r_2, r_3\} \in R.\\ &\neg P(s, \mathtt{network\_state}, r_1)\\ &\vee (P(s, \mathtt{network\_state}, r_2) \wedge\\ &\quad P(s, \mathtt{network}, r_3))\end{aligned}$$

Informally, this formula shows a rights dependence between access to network_state and network.

5.3. Applications Invariants

Invariant 7: *"An application can only receive SMS notifications from trusted system components."*

Description: Any application can broadcast an Intent; however, some Intents should only originate from trusted system components. This invariant protects an application by ensuring only the system can send it an SMS message. We assume PhoneApp is the application trusted to send SMS messages. This invariant generalizes to any notification assumed to be broadcasted by a trusted system component.

Logic: For each such Broadcast Receiver $\mathtt{BR}_i$ subscribed to the SMS_RECEIVED action string (known by the application developer), $\mathtt{inv}_{7i}$ must be true for all applications $s \in S$ installed on the system.

$$\mathtt{inv}_{7i}(s) = \forall r_1 \in R, \exists r_2 \in R.\ \neg P(s, \mathtt{BR}_i, r_1) \vee (P(s, \mathtt{BR}_i, r_2) \wedge (s \equiv \mathtt{PhoneApp}))$$

Informally, this formula states that if s can broadcast an Intent to $\mathtt{BR}_i$, then s must be PhoneApp. Note, dynamic Broadcast Receivers, require either source code analysis or developer aid, see Section 7.1.

Invariant 8: *"An application can only receive location updates from trusted system components."*

Description: The SDK provides two methods for location updates: interval (i.e., time or distance) and proximity. Beginning with the v0.9r1 SDK release, interval-based updates use a callback mechanism with the location manager running with the system_server. Proximity updates arrive via an Intent broadcasted to a Broadcast Receiver defined in a PendingIntent. The invariant protecting an application from forged location updates need only consider proximity updates, as the application initiates the ICC for interval updates.

Logic: For each Broadcast Receiver $\mathtt{BR_i}$ used for location updates (known by the application developer), $\mathtt{invariant_{8i}}$ must be true for all applications $s \in S$ installed on the system.

$$\mathtt{inv}_{8i}(s) = \forall r_1 \in R, \exists r_2 \in R.\ \neg P(s, \mathtt{BR}_i, r_1) \vee (P(s, \mathtt{BR}_i, r_2) \wedge (s \equiv \mathtt{system\_server}))$$

Similar to Invariant 7, this states that if s can broadcast an Intent to $\mathtt{BR}_i$, then s must be system_server.

6. Security Evaluation

The policy invariants described in Section 5 define illustrative, but realistic, security requirements for a mobile phone. In this section, we show that Android's existing security framework is insufficient by evaluating the Android framework and SDK bundled applications against our invariants. These applications represent a best effort of ad hoc permission assignment to protect core mobile phone functionality such as voice and SMS services, and upon evaluation of these invariants we identified multiple vulnerabilities arising from policy misconfiguration. The analysis described herein was performed on the v0.9r1 SDK release; however, the presented results hold for the v1.0r1 release. The remainder of this section describes our analysis tool, interesting test results, and proof-of-concept malware exploiting discovered flaws.

6.1. Implementation

We developed Kirin as an Android application to evaluate invariants using only application packages (`.apk` files) as input. Our implementation currently only performs invariant evaluation on the phone. Kirin follows the installation flow presented in Figure 5, and with the release of Android's system source code, we are currently implementing tighter integration with the package installation system.

Kirin first extracts policy knowledge from the target application package using the `PackageManager` and `PackageParser` APIs included in the Android SDK and stores the result as Prolog facts (e.g., **`has_perm`**(), **`requires`**(), **`contains`**()) in a temporary file. It then performs the preprocessing steps described in Section 4.2 and merges previously derived Prolog facts corresponding to the framework and applications required for basic phone operation (i.e., `Phone`, `TelephonyProvider`, `DownloadProvider`, `Laucher`, `SdkSetup`, `SettingsProvider`). Finally, Kirin uses the XSB [54] Prolog engine, which we recompiled for the ARM-based Android environment, to test each of our invariants. In all cases, the invariants in Section 5 were trivially encoded into Prolog. See the appendix for a screenshot of Kirin.

6.2. Invariant Evaluation

The provided system applications represent most, but not all, of the functionality considered in our eight sample invariants. Table 1 presents the experimental results with interesting failures. We omit the results of many invariants, because the tested applications are either installed in `/system/apps` or do not use the specialized interfaces, and therefore pass trivially. Additionally, we added two example applications to demonstrate a subtle programming error related to the location API. We now discuss the origins of failure.

Unchecked Interfaces – Surprisingly, all applications fail Invariant 1, which ensures applications request the `CALL_PHONE` permission in order to make an outgoing phone call. All applications fail, because the Phone application (included in the base policy knowledge) contains an Activity that will automatically dial out if provided a telephone number as a parameter, but does not specify an access permission. While this vulnerability is easily mitigated by adding an access permission, it demonstrates the utility of Kirin: *identifying sensitive interfaces is easier than ensuring proper permission settings.*

Controlling Dangerous Permissions – All applications that simultaneously contain a dangerous permission and define a main Activity fail Invariant 2. The failure results, because the main Activity of an application *must not* contain an access permission, otherwise it can never be started. While we could model this proper execution by defining a new reserved right ε_{launch} for components launched from the desktop, the discovery indicates that *applications containing dangerous permissions may be maliciously controlled via start parameters.* We expect source code analysis to help identify Activities not consuming parameters.

Intent Origin – The MMS application, which handles all SMS and MMS messages, fails Invariant 7. Further investigation indicates no access permission on its SMS Broadcast Receiver. Hence, Kirin's preprocessor assigns the ε right, indicating any application can broadcast Intents to the MMS application. This implies an unprivileged malicious application can forge SMS messages. Fortunately, this specific vulnerability can be mitigated by defining a new permission for broadcasting SMS Intents. However, this vulnerability more generally impacts all system broadcasted Intents, indicating a larger concern: *applications cannot trust data broadcasted to system defined action strings.*

Location API – The Maps application only uses interval location updates, therefore it trivially passes Invariant 8. Recent releases of the SDK broadcast proximity Intents using the special *PendingIntent* class, which allows a location client to register for an explicitly addressed broadcast to a private component. If done correctly, proximity updates are secure; however, subtle inference rules can make a component public and therefore subject to forgery. Developers first introduced to the previous M5 SDK are at greater risk of poor programming practices affecting this vulnerability. In Table 1, ProximityClientA was intentionally developed

Table 1: Applications Presenting Invariant Failures

	System Applications				Example Applications	
Invariant	Browser	Maps	Mms	Music	ProximityClientA	ProximityClientB
1	F	F	F	F	-	-
2	F	F	F	F	-	-
7	P	P	F	P	-	-
8	-	P*	-	-	F	P

P = Pass; F = Fail; - = Not tested; P* = Pass, but proximity alert not used

with poor practices, whereas ProximityClientB was developed using best practices. The invariant test results indicate that *Kirin can help developers identify poor programming practices leading to vulnerabilities.*

6.3. Malicious Applications

We developed three applications that exploit the vulnerabilities identified by Kirin. A system that passes our sample invariants is not susceptible to these attacks. However, each attack has been verified on the SDK emulator. See the appendix for screenshots.

Making Unprivileged Phone Calls – The unprotected component in the Phone application allows an unprivileged application to make phone calls to arbitrary phone numbers (however, emergency numbers, e.g., 911, still require the `CALL_PRIVILEGED` permission). By starting an Activity for the `CALL` action string and specifying `tel:<phone_number>` as a data parameter, the phone will dial without user confirmation. Our proof-of-concept application waits until the screen is blanked before connecting the call in an effort to hide the event from the user. The act of making a call re-enables the screen, and the call must be remotely disconnected; however, once the call is connected, eavesdropping can proceed until the user becomes cognizant of the event.

Forging SMS – The lack of a standard convention for protecting Broadcast Receivers subscribed to the `SMS_RECEIVED` Intent leaves all SMS listeners vulnerable to a forging attack. Our unprivileged attack application broadcasts a standard GSM SMS PDU data structure [20], [21] to the `SMS_RECEIVED` action string, which is successfully received by the MMS application. As such, our attack application can forge messages from emergency alert systems or for specific applications (e.g., the Grey access control system [5]).

Forging Location – Our last application attacks the vulnerable ProximityClientA application constructed for Section 6.2. By knowing either the Broadcast Receiver's component name, or the action string to which it subscribes, our attack program can forge a proximity alert indicating that the phone has either entered or exited an area. This type of forgery is especially dangerous when using proximity updates for security decisions, e.g., physical access control.

7. Limitations and Future Extensions

7.1. Extracting Security Policy

Kirin is currently limited to knowledge obtainable from the application package metadata, e.g., the manifest file. Therefore, dynamically created Broadcast Receivers, which are not specified in the manifest file, are not automatically analyzed. Note that our invariants involving Broadcast Receivers were classified as "protecting the application," therefore, we can expect aid from the developer. While we are working on source code analysis additions to automate detection, we propose extending the Manifest file to include specifications for dynamically created Broadcast Receivers. Additionally, we are working on an extension to identify broadcasted Intents, which, as described in Section 4.2, are modeled as objects. Such source code analysis tools could also automatically populate the manifest file for developers, ensuring it contains a policy upper bound for use by Kirin or similar analysis tools. Note that Android would require changes to enforce these new upper bounds; however, the open source nature of Android makes it amenable to such modification.

In addition to the manifest file, the Android SDK allows developers to specify arbitrary permission checks within the program code, e.g., "Service Hooks." As described in Section 2.2, while we expect Service hooks to follow straightforward patterns, extraction of arbitrary policy from source code may be intractable, or even undecidable. As an alternative approach, we propose modifying the manifest file to include a per-interface policy specification, allowing the developer to separate policy and mechanism.

7.2. Information Flow

Mandatory Access Control (MAC) mechanisms, e.g., SELinux [39], commonly support information flow security policies. For example, in Invariant 5, it may have been more appropriate to define logic that

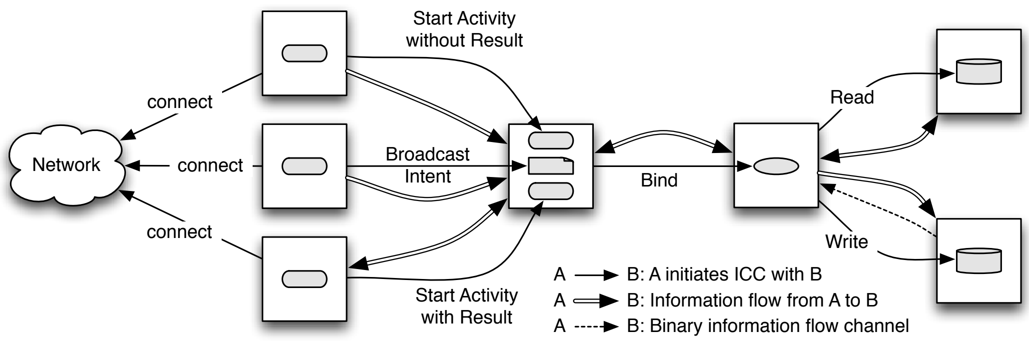

Start Activity without Result
connect
Network
connect
Broadcast Intent
connect
Bind
Read
Write
Start Activity with Result
A → B: A initiates ICC with B
A ⇒ B: Information flow from A to B
A ----> B: Binary information flow channel

Figure 7: Information flow implications of ICC establishment.

ensures recorded audio cannot reach an application with network access through any number of hops. Unfortunately, the Android architecture itself makes useful information flow policies difficult to create. As shown in Figure 7, considering opaque ICC establishment often results in bidirectional flows between components (the double arrows). Therefore, many applications may reside in symmetric security classes. This property is further exacerbated by implicitly open Activities used to launch an application, i.e., information can flow into any application with a user interface. The inclusion of source code analysis to determine flow directionality or trusted filters may make such polices practical. We leave a deeper analysis of the potential for information flow policies in Android for future work.

8. Related Work

Authorization decisions traditionally stem from the Access Matrix [30], which specifies if a subject (e.g., process) can access some object (e.g., file) for some action (e.g., read). HRU [26] demonstrate a logic for managing the access matrix for discretionary access control. While conceptually, the access matrix is the fundamental building block for access control systems, sparse matrices and policy specification often motivate alternate representations. Logics that describe distributed authorization [1], Trust Management [6], [7], [18], or polices in general [25] enable formal proofs to reason about the security of a system, and many works define access control logics based on Datalog [17], [28], [29], [31], [32], [33], a subset of Prolog. While Android's distributed component infrastructure and application specified policies tangentially relate to these distributed authorization systems, its core system embodies a reference monitor [2], allowing globally aware policies. With centralized enforcement, operating system reference monitors provide a framework for enforcing mandatory access control capable of expressing information flow polices [15].

Software installation represents a broad challenge of systems security research, with significant focus on containing Trojans after the fact [22], [39], [43]. Code signing provides a mechanism to authenticate software origin; however, it does not express how programs are expected to interact with the system. Rueda et al. [44] consider applications with predefined policies; however, this work is limited to verifying the compliance of a small set of trusted utilities written in a security typed language, rather than mediating software installation a whole. In the mobile spectrum, Trojans and viruses are an increasing problem [46], with virus propagation through multiple vectors [9], which has lead to literature describing various detection techniques [8], [12], [47]. The state of secure software installation on mobile phones resembles that of commodity operating systems, with application signing [35], [48] providing the most common malware mediation. However, more advanced mechanisms are emerging. Mobile phone security techniques based on SELinux and Trusted Platform Modules are increasingly common [38], [55], and Desmet et al. [16] describe an architecture for mobile phone application distribution based on "security by contract." The latter has similar motivations as our installer, but is targeted at the Windows Mobile framework.

9. Conclusion

The move from closed to open mobile phone systems requires attention. While these phones are subject to the same types of Trojans and viruses as general purpose systems, tight integration with fragile cellular networks and user privacy significantly raises security stakes. In this paper, we proposed Kirin as an alternate application installer and security framework for the Google Android mobile phone platform that allows the phone to enforce *policy invariants* that transcend applications. Our goals presented many challenges in terms of both modeling the existing security mechanisms and acquiring appropriate policy primitives. However, once in place, the formal model allowed automated analysis to identify insecure policy configurations leaving the phone vulnerable to attack. While our current implementation relies on information residing in manifest files, future work will apply source code analysis to more accurately model application interaction, and ultimately define a more secure method of installing applications on mobile platforms.

Acknowledgements

We would like to thank Stephen McLaughlin, Sandra Rueda, and the rest of the SIIS lab, as well as and John Hannan for their insightful comments during the writing of this paper.

References

[1] M. Abadi, M. Burrows, and B. Lampson. A Calculus for Access Control in Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[2] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, The Mitre Corporation, Air Force Systems Division, Hanscom AFB, Badford, MA, October 1972.

[3] Apple Inc. AppStore. http://www.apple.com/iphone/appstore/. Accessed September 2008.

[4] Apple Inc. iPhone DevCenter. http://developer.apple.com/iphone/. Accessed September 2008.

[5] L. Bauer, S. Garris, J. M. McCune, M. K. Reiter, J. Rouse, and P. Rutenbar. Device-enabled authorization in the grey system. In *Proceedings of the 8th Invormation Security Conference*, pages 431–445, September 2005.

[6] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 164–173, May 1996.

[7] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the policymaker trust management system. In *Proceedings of Financial Cryptography*, pages 254–274, February 1998.

[8] A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral Detection of Malware on Mobile Handsets. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2008.

[9] A. Bose and K. G. Shin. On Mobile Viruses Exploiting Messaging and Bluetooth Services. In *Proceedings of the International Conference on Security and Privacy in Communication Networks (SecureComm)*, August 2006.

[10] J. Boudreau. Q&A with Peter Chou, chief executive and co-founder-HTC. Mercury News, October 2008. http://www.mercurynews.com/businessheadlines/ci_10799172.

[11] D. F. Brewer and M. J. Nash. The Chinese Wall Security Policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 206–214, May 1989.

[12] J. Cheng, S. H. Wong, H. Yang, and S. Lu. SmartSiren: Virus Detection and Alert for Smartphones. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2007.

[13] E. Chu. Android Market: a user-driven content distribution system. http://android-developers.blogspot.com/2008/08/android-market-user-driven-content.html, August 28, 2008.

[14] S. Corp. Security Updates: SymbOS.Locknut. http://securityresponse1.symantec.com/sarc/sarc.nsf/html/symbos.locknut.html, July 4, 2006.

[15] D. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[16] 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 the ACM Computer Security Architecture Workshop*, pages 19–28, November 2007.

[17] J. DeTreville. Binder, a Logic-Based Security Language. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 105–113, 2002.

[18] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory. IETF RFC 2693, September 1999.

[19] P. Elmer-DeWitt. iPhone: Big trouble in the App Store. http://apple20.blogs.fortune.cnn.com/2008/09/14/iphone-big-trouble-in-the-app-store/, September 14, 2008.

[20] European Telecommunications Standards Institute. Digital cellular telecommunications system (phase 2+); alphabets and language-specific information. (ETS 100 900, GSM 03.38 version 7.2.0 Release 1998), July 1999.

[21] European Telecommunications Standards Institute. Digital cellular telecommunications system (phase 2+); technical realization of the short message service (sms); point-to-point (pp). (TS 100 901, GSM 03.40 version 7.3.0 Release 1998), November 1999.

[22] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the USENIX Security Symposium*, 1996.

[23] Google. Android - An Open Handset Alliance Project. http://code.google.com/android/. Accessed September 2008.

[24] Google. Android Application Model: Applications, Tasks, Processes, and Threads. http://code.google.com/android/intro/appmodel.html. Accessed September 2008.

[25] J. Halpern and V. Weissman. Using First-Order Logic to Reason about Policies. *ACM Transactions on Information and Systems Security*, 11(4), July 2008.

[26] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[27] C. Jade. iPhone Market Share Surges in August. Ars Technica, September 2008. http://arstechnica.com/journals/apple.ars/2008/09/

01/iphone-market-shares-surges-in-august.

[28] S. Jajodia, P. Samarati, M. L. Sapino, and V. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2):214–260, June 2001.

[29] T. Jim. SD3: a trust management system with certified evaluation. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 106–115, May 2001.

[30] B. Lampson. Protection. In *Proceedings of the Fifth Princeton Symposium of Information Science and Systems*, pages 437–443, March 1971.

[31] N. Li, B. Grosof, and J. Feigenbaum. Delegation Logic: A Logic-based Approach to Distributed Authorization. *ACM Transactions on Information and System Security*, 6(1):128–171, February 2003.

[32] N. Li and J. Mitchell. Understanding SPKI/SDSI using first-order logic. *International Journal of Information Security*, 5(1):48–64, January 2006.

[33] N. Li, J. Mitchell, and W. Winsborough. Design of a Role-Based Trust Management Framework. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 114–130, May 2002.

[34] Microsoft. Mobile2Market. http://msdn.microsoft.com/en-us/windowsmobile/bb250547.aspx. Accessed September 2008.

[35] Microsoft. Windows Mobile Powered Device Security Model. http://msdn.microsoft.com/en-us/library/bb416353.aspx, August 2008.

[36] Microsoft Corporation. COM: Component Object Model Technologies. http://www.microsoft.com/com/. Accessed September 2008.

[37] D. N. MSN UK. Microsoft to rival Apple and Google with Skymarket app store. http://tech.uk.msn.com/news/article.aspx?cp-documentid=9419196, September 1, 2008.

[38] D. Muthukumaran, A. Sawani, J. Schiffman, B. M. Jung, and T. Jaeger. Measuring integrity on mobile phone systems. In *Proceedings of the ACM Symposium on Access Control Models and Technologies*, pages 155–164, June 2008.

[39] National Security Agency. Security-enhanced Linux (SELinux). http://www.nsa.gov/selinux.

[40] OpenBinder: And Open-Source System Component Framework. http://www.open-binder.org. Accessed July 2008.

[41] Openmoko. http://www.openmoko.com. Accessed July 2008.

[42] S. B. PDABlast Staff. Skulls Trojan Virus Hits Symbian Phones. http://symbian.pdablast.com/articles/2004/11/20041122-Skulls-Trojan-Virus-Hits.html, November 22, 2004.

[43] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–271, Washington, D.C., Aug. 2003.

[44] S. Rueda, D. King, and T. Jaeger. Verifying compliance of trusted programs. In *Proceedings of the USENIX Security Symposium*, pages 321–334, August 2008.

[45] J. Saltzer and M. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), September 1975.

[46] A.-D. Schmidt and S. Albayrak. Malicious Software for Smartphones. Technical Report TUB-DAI 02/08-01, DAI-Labor, February 10, 2008.

[47] A.-D. Schmidt, F. Peters, F. Lamour, and S. Albayrak. Monitoring Smartphones for Anomaly Detection. In *Proceedings of the International Conference on Mobile Wireless Middleware, Operating Systems, and Applications (Mobilware)*, February 2008.

[48] Symbian Ltd. Symbian signed. https://www.symbiansigned.com. Accessed September 2008.

[49] T-Mobile. T-Mobile devPartner Community - Beta. http://developer.t-mobile.com/. Accessed September 2008.

[50] The Object Management Group. Common Object Request Broker Architecture: Core Specification. http://www.omg.org/docs/formal/04-03-12.pdf, March 2004.

[51] P. Traynor, W. Enck, P. McDaniel, and T. L. Porta. Exploiting Open Functionality in SMS-Capable Cellular Networks. *Journal of Computer Security*, 2008. *to appear*.

[52] P. Traynor, W. Enck, P. McDaniel, and T. L. Porta. Mitigating Attacks on Open Functionality in SMS-Capable Cellular Networks. *IEEE/ACM Transactions on Networking (TON)*, 2008. *to appear*.

[53] M. P. Viruses. Brador on Windows CE for Pocket PCs. http://www.mobilephoneviruses.com/mobile-virus/brador, October 2006.

[54] Xsb. http://xsb.sourceforge.net, August 2008.

[55] 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*, pages 7–14, November 2007.

Appendix A. Prolog Example

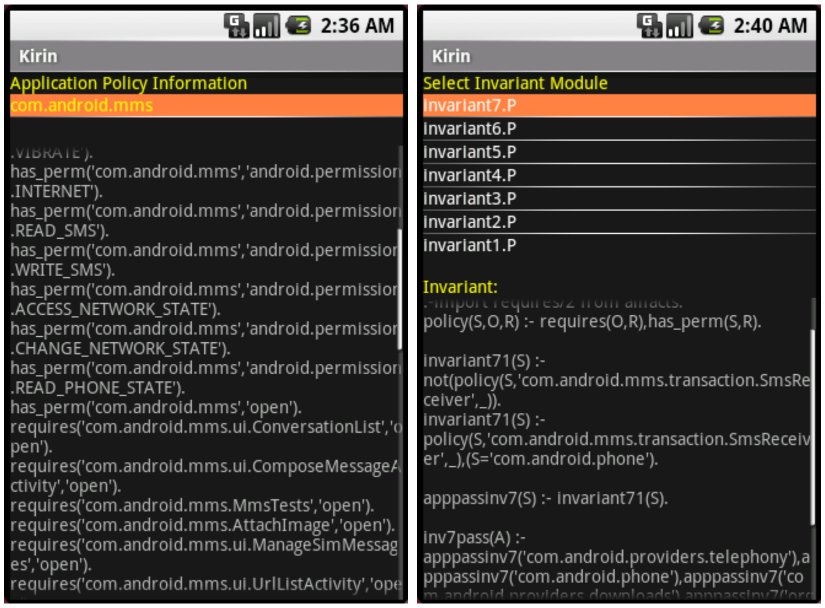
The following is a Prolog encoding of the invariant patterns defined in Section 4.3. For example, `MapsApp` does not pass $\mathtt{pat}_2$ because it possesses both the `READ_CONTACTS` and `INTERNET` permissions.

```
has_perm('ContactsApp','READ_CONTACTS').
has_perm('ContactsApp','WRITE_CONTACTS').
has_perm('MapsApp','READ_CONTACTS').
has_perm('MapsApp','INTERNET').
requires('ContactsProvider_r','READ_CONTACTS').
requires('ContactsProvider_w','WRITE_CONTACTS').
requires('network','INTERNET').

policy(S,O,R) :- requires(O,R),has_perm(S,R).
pattern1(S)   :- policy(S,'ContactsProvider_w',_).
pattern2(S)   :- not(policy(S,'ContactsProvider_r',_)).
pattern2(S)   :- policy(S,'ContactsProvider_r',_),not(policy(S,'network',_)).
pattern3(S)   :- not(policy(S,'ContactsProvider_w',_)).
pattern3(S)   :- policy(S,'ContactsProvider_w',_),policy(S,'ContactsProvider_r',_).

| ?- pattern1('ContactsApp').
yes
| ?- pattern1('MapsApp').
no
| ?- pattern2('MapsApp').
no
| ?- pattern2('ContactsApp').
yes
| ?- pattern3('ContactsApp').
yes
```

Appendix B. Screenshots



```
.VIBRATE').
has_perm('com.android.mms','android.permission
.INTERNET').
has_perm('com.android.mms','android.permission
.READ_SMS').
has_perm('com.android.mms','android.permission
.WRITE_SMS').
has_perm('com.android.mms','android.permission
.ACCESS_NETWORK_STATE').
has_perm('com.android.mms','android.permission
.CHANGE_NETWORK_STATE').
has_perm('com.android.mms','android.permission
.READ_PHONE_STATE').
has_perm('com.android.mms','open').
requires('com.android.mms.ui.ConversationList','o
pen').
requires('com.android.mms.ui.ComposeMessageA
ctivity','open').
requires('com.android.mms.MmsTests','open').
requires('com.android.mms.AttachImage','open').
requires('com.android.mms.ui.ManageSimMessag
es','open').
requires('com.android.mms.ui.UrlListActivity','ope
```

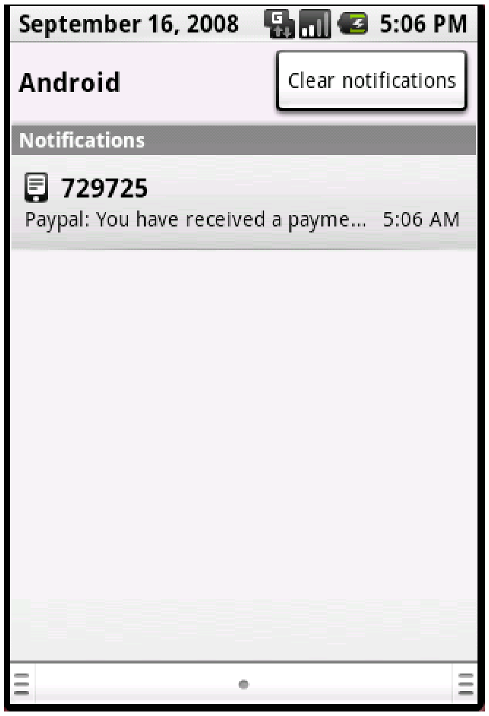
```
:-import requires/2 from allfacts.
policy(S,O,R) :- requires(O,R),has_perm(S,R).

invariant71(S) :-
not(policy(S,'com.android.mms.transaction.SmsRe
ceiver',_)).
invariant71(S) :-
policy(S,'com.android.mms.transaction.SmsReceiv
er',_),(S='com.android.phone').

apppassinv7(S) :- invariant71(S).

inv7pass(A) :-
apppassinv7('com.android.providers.telephony'),a
pppassinv7('com.android.phone'),apppassinv7('co
m.android.providers.downloads'),apppassinv7('or
```

Figure 8: Kirin extracts security-related facts from the application as shown on the left figure. These facts are merged with the existing facts which represent the current security state of the phone. The right figure shows an example Prolog encoding of a security invariants to be tested during the installation.


September 16, 2008 5:06 PM

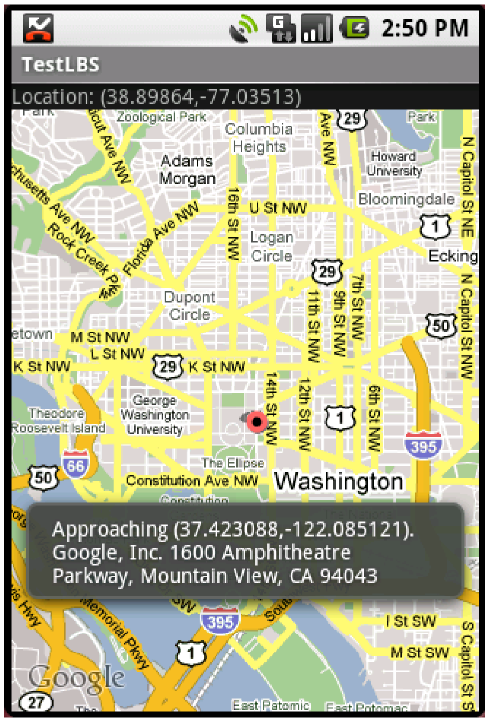
Android

Notifications

729725

Paypal: You have received a payme... 5:06 AM

Figure 9: SMS Attack - Our malicious SMS forging application sends a forged Paypal notification message, which cannot be distinguished from legitimate SMS messages.


2:50 PM

TestLBS

Location: (38.89864,-77.03513)

Approaching (37.423088,-122.085121). Google, Inc. 1600 Amphitheatre Parkway, Mountain View, CA 94043

Figure 10: Proximity Update Attack - Our malicious application sends forged proximity alert to confuse the user that the device is approaching Google Inc. in CA while it is in Washington DC.

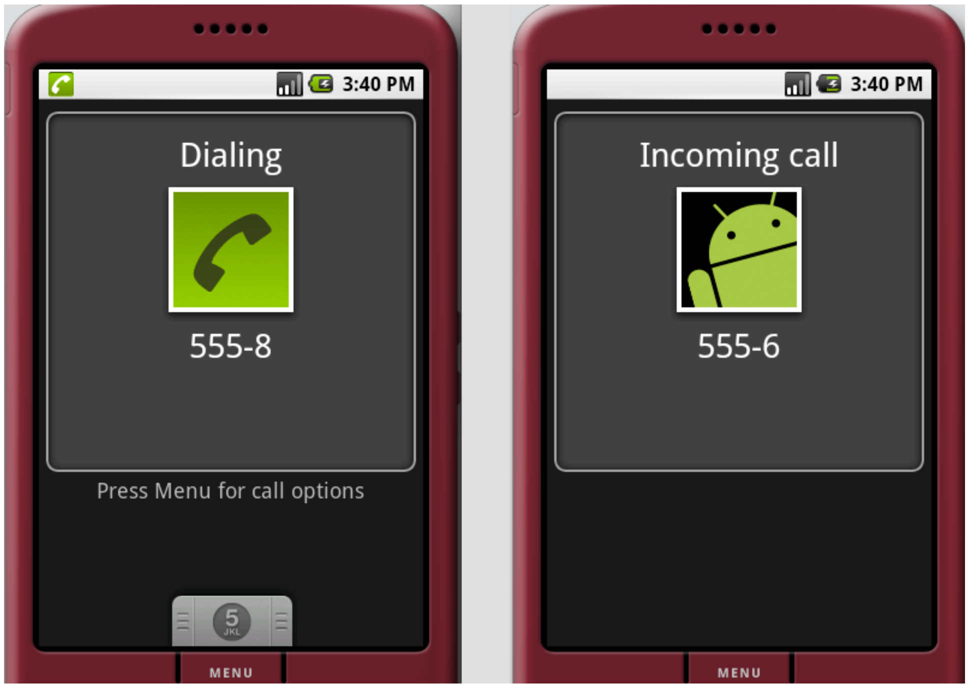


Figure 11: Voice Call Attack - Without any permission assigned to it, our unprivileged caller can make an outgoing voice call to another device. Note that each Android emulator uses the port number as the device's phone number.