Architecture Modeling and Analysis of Security in Android Systems

Bradley Schmerl[1], Jeff Gennari[1], Alireza Sadeghi[2], Hamid Bagheri[2], Sam Malek[2], Javier Cámara[1], and David Garlan[1]

[1] Institute for Software Research, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA USA 15221

[2] School of Information and Computer Sciences, University of California, Irvine, Irvine, CA, USA 92697

Abstract. Software architecture modeling is important for analyzing system quality attributes, particularly security. However, such analyses often assume that the architecture is completely known in advance. In many modern domains, especially those that use plugin-based frameworks, it is not possible to have such a complete model because the software system continuously changes. The Android mobile operating system is one such framework, where users can install and uninstall apps at run time. We need ways to model and analyze such architectures that strike a balance between supporting the dynamism of the underlying platforms and enabling analysis, particularly throughout a system's lifetime. In this paper, we describe a formal architecture style that captures the modifiable architectures of Android systems, and that supports security analysis as a system evolves. We illustrate the use of the style with two security analyses: a predicate-based approach defined over architectural structure that can detect some common security vulnerabilities, and inter-app permission leakage determined by model checking. We also show how the evolving architecture of an Android device can be obtained by analysis of the apps on a device, meaning that the architecture can be amenable for use throughout the system's lifetime.

1 Introduction

Software architecture modeling is an important tool for early analysis of quality attributes. Architecture analysis of run-time quality attributes such as performance, availability, and reliability can increase confidence at design time that quality goals will be met during implementation. Component and connector view architectures are especially important to reason about the desired run-time qualities of the system.

Architecture analysis of security involves understanding the information flow through an architecture to uncover security related issues such as information leakage, privilege escalation, and spoofing. Many of these run time analyses assume the existence of complete architectures of the system being analyzed, and in the case of security analysis, knowledge of the entire information flow of the system.

However, in many modern systems, architectures can evolve and change at run time, and so new paths of communication, and thus new vulnerabilities, can be introduced into these systems. A critical example of this is software frameworks, which are used in the commercial sector and increasingly in the defense sector as well. Frameworks offer a means for achieving composition and reuse at scale — frameworks can be extended with plugins during use. Examples of such frameworks include mobile device software, web browser extensions, and programming environments. The mobile device arena, in particular the Android framework, is an interesting case. The Android framework provides flexible communication between apps (plugins that use the framework) that allows other apps to provide alternative core functionality (such as browsing, SMS, or email) or to tailor other parts of the user experience. However, this flexibility can also be exploited by malicious apps for nefarious purposes. We need ways to analyze the architectures of these systems, in particular for security properties.

Like many of these frameworks, the architectures of the software of Android devices exhibit a number of challenges when it comes to modeling and analysis of security properties: (1) the system architectures evolve as new apps are installed, activated, and used together; (2) the architectures of each app, while conforming to a structural style, are constructed by independent parties, with differing motivations and tradeoffs, (3) there are no common goals for a particular device.

This means that security needs to be reanalyzed as the system changes. In particular, we need to be able to do analysis over a good model that is abstract enough to be tractable, yet detailed enough to support analysis. And so there is a question of how to specify the new evolved architecture, and at what level of abstraction. We also need a way to derive the architecture from the system itself, so that throughout the software's lifecycle we can perform the analysis when we know all the communication paths.

In this paper, we describe an architecture style for Android that supports run-time analysis of security. We show how instances of this style can be derived from Android apps to specify an up-to-date architecture of the entire software on an Android device. We also give examples of two kinds of security analysis that is supported for this style — constraint-based analysis that detects the presence of a category of threats commonly known as STRIDE [21], and a model-checking approach that determines potentially vulnerable communication pathways among apps that may result in leakage of information and permissions.

This paper is organized as follows: In Section 2 we introduce Android, and discuss work on architecture modeling of Android and architecture-based security analysis. Building on this, we define the requirements for an architecture style for Android security analysis in Section 3 and then describe the Acme architecture style in Section 4. We describe how to automatically derive instances of this style from Android apps in Section 5. Section 6 describes two security analyses using this architectural abstraction. In Section 7, we show how long it takes to discover the architecture of a number of differently sized apps available in the play store. We conclude with discussion and future work in Section 8.

2 Background and Related Work

In this section, we first provide an overview of Android to help the reader follow the discussions that ensue. We then provide an overview of the prior work in architectural modeling and analysis, particularly with respect to the security of Android.

2.1 Introduction to Android

Android is a popular operating system for mobile devices, like phones, tablets, etc. It is deployed on a diverse set of hardware, and can be customized by companies to provide additional features. Android is designed to allow programs, known as apps, to be installed on the device by end users. From an operating system perspective, Android provides apps with communication mechanisms and access to underlying device hardware and services, such as telephony features.[3] Furthermore, it allows end-user extension in the form of installing additional apps that are provided by third parties. The provision of explicit communication channels between apps allows for rich app ecosystems to emerge. Apps can use standard apps for activities such as web browsing, mapping, telephony, messaging, etc., or they can be flexible and allow third party apps to handle these activities. Because security is a concern in Android, apps are sandboxed from each other (using the Unix account access control where every app has its own account), and can only communicate through the mechanisms provided by Android.

An app in Android specifies in a manifest file what activities and other components comprise it. In this manifest, activities further specify the patterns of messages that they can process. Furthermore, apps specify the permissions that they require that need to be granted by users when they install the apps.[4] Activities in an app communicate by sending and receiving messages, called *intents*. These intents can be sent either to other activities within the app, or to activities that belong to other apps. There are two forms of intents: explicit and implicit. For explicit intents, activities specify the activity that should receive and process the intent – which can be either inside the app or in another app. For implicit intents, an activity does not specify the recipient. Instead, Android conducts intent resolution that matches the intent with intent patterns specified by activities. So, for example, an activity can request that a web page be displayed, but can allow that web page to be displayed by third party browsing apps that may be unknown at the time the requesting app is developed.

While intents provide a great deal of flexibility, they are also the source of a number of security vulnerabilities such as intent spoofing, privilege escalation, and unauthorized intent receipt [9]. To some degree, these vulnerabilities can be uncovered by analyzing apps and performing static analysis to see how intents are used, what checks are made on senders and receivers of intents, and so on [20]. However, Android is an extendable platform that allows users to dynamically download, update, and delete apps that makes a full static analysis impossible.

2.2 Security Architecture Modeling and Analysis

Many security vulnerabilities in Android result from unexpected interactions between components. Architecture is the right place to identify these interactions because these interactions can be evaluated at the system level. Many of the communication pathways of interest are specified at the component level within the manifest definition of the

[3] https://developer.android.com/

[4] The most recent version of Android, Marshmallow, has a more dynamic form of permission granting, which allows permissions to be granted as they are needed dynamically by the app. This paper discusses the Lollipop version of Android.

app, or can be extracted by analyzing calls in its bytecode (described in Section **??**). Therefore, we can focus our analysis for security at the architecture level - analyzing at the level of components and interactions.

Specialized ADLs geared towards security analysis exist. These tend to focus on specific security properties, such as access control [19,11]. In [2] UML OCL-type constraints are used to specify constraints that uncover threats specified by the Common Attack Pattern Enumeration and Classification (CAPEC)[5] and provide tool support for security risk analysis during the system design phase using system architecture and design models.

The key point of these architectural security analyses is that the communication pathways need to be represented at the architecture level, along with the security relevant properties needed for analysis. However, all of this work relates to security design, and so there is an assumption that a complete architecture is analyzed. Furthermore, the approaches discussed rely on developers to implement the systems according to the architecture. To enable these kinds of analysis on Android requires being able to extract the properties relevant to security from Android apps.

In [3], the authors study the extent to which Android apps employ architectural concepts in practice. This study provides a characterization of architectural principles found in the Android ecosystem, supported with mining the reverse-engineered architecture of hundreds of Android apps in several app repositories. We build on this work to provide automated architectural extraction from Android devices.

SEPAR [6] provides an automatic scheme for formal synthesis of Android inter-component security policies, allowing end-users to safeguard the apps installed on their device from inter-component vulnerabilities. It relies on a constraint solver to synthesize possible security exploits, from which fine-grained security policies are derived. Such fine-grained, yet system-specific, policies can then be enforced at run time to protect a given device.

Bagheri et al. conduct a bounded verification of the Android permission protocol modeled in terms of architectural-level operations [4]. The results of this study reveal a number of flaws in the permission protocol that cause serious security defects, in some cases allowing the attacker to entirely bypass the Android permission checks.

SECORIA [1] provides security analysis for architectures and conformance for systems with an underlying object-oriented implementation. Through static analysis, a data flow architecture of a system is constructed, as in instance of a an data flow architecture style defined in Acme [17]. Components are assigned a trust level and data read and write permissions are specified on data stores. Security constraints particular to a software systems (such as that "Access to the key vault [. . .] should be granted to only security officers and the cryptographic engine") are captured as Acme constraints. In [15], the DFD style was extended with constraints for analyzing a subset of the STRIDE vulnerabilities. We show in Section 6.2 how this latter approach can be applied to analyze vulnerabilities in Android.

[5] http://capec.mitre.org

3 Modeling Requirements for Android

To evaluate the security of Android apps, the core Android architectural structures need to be represented in a modeling language. Android app component types, such as activities, services, and content providers form the building blocks for all apps. Each Android component type possesses properties that are critical for security assessment. For example, activities can be designated as "exported" if they can be referenced outside of the app to which they belong. Exported activities are a common source of security vulnerabilities, thus a security-focused architectural model must include information about whether an activity is exported. Android apps are distinct, yet they share many commonalities necessary for app creation and interaction. A major consequence of this design is that boundaries between apps are loosely defined and enforced. To identify and evaluate potential security issues that emerge from app interaction on a device, all apps and their connections deployed on the device must be made explicit in the architecture. Furthermore, because apps can be updated, installed, and removed during the lifetime of the device, the architecture model must be flexible and easy to modify.

Since a significant number of Android security issues arise from unexpected interactions between apps, modeling communication pathways between apps on a device is perhaps the most critical requirement for security analysis. At the device level, each individual app is essentially a subsystem that operates in the context of a larger, device-wide ensemble. Apps are often designed to rely on other apps, many of which may not be known at design time. For instance, if an app needs a mapping service, it does not necessarily know which specific mapping service will be available at run time. An app can be reasonably secure in isolation, but when evaluated in the presence of other apps, it may contribute to critical security vulnerabilities.

Android's intent passing system provides a common communication mechanism to simplify inter-app communication. Android supports many intent passing modes, such as asynchronous and synchronous delivery. However, Android's intent passing system includes additional semantics that can have different security implications. For example, whether an intent is delivered to one specific component or broadcasted to all components. Differences in intent passing semantics need to be made explicit in an Android architecture style to enable analysis of common security issues that rely on certain types of communication, such as intent or activity spoofing. Android app components and connectors are organized in apps by configuring them via a manifest file. The boundaries set in the manifest creates an important trust boundary and must be explicit to evaluate data flowing in to, or out of, an app. To be complete, the architectural style must support component-to-component interaction and inter-app communication.

Android permissions are another core mechanism used to prevent security-related issues. Given the pervasive intent-based communication system, it is left to permissions to control access and information flow between components. Android supports a wide array of core permissions and provides ways to add new permissions. Due in part to the nature of Android permission management, many security issues result from components with insufficient privileges gaining access to privileged components and system resources. The architecture modeling language must support Android privileges. Identifying security vulnerabilities often involves determining whether permissions can be subverted. Thus, permissions must be attached to various resources in a way that allows them to be analyzed.

4 An Android Architecture Style

Based on the requirements above, we define a formal architectural model of the Android framework in order to have a basis for modeling the structures and constraints in Android, and to permit analysis of security vulnerabilities and exploits. To do this, we have developed an architectural style in Acme [17], that represents intent interactions and permissions in Android.

All components types specify a property that indicates the class that implements them and the permission needed to access them, as well as whether they are exported (or public) to other applications. The types of components are:

AcvtivityT: This component type specifies an activity within an app. Activities represent components in an app that have a user interface. Communicating with that activity involves instantiating this user interface. Specified with the activity are the intent patterns that it understands and can process, the intents that it sends, in addition to the the services and resources that it accesses. These latter communications are all represented by distinct port types, as described below.

ServiceT: This component type specifies a service within an app. According to the standard Android description, “a Service is an application component that can perform long-running operations in the background and does not provide a user interface. Another application component can start a service and it will continue to run in the background even if the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). For example, a service might handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.”[6] Services have ports that specify the interfaces they provide and the services they use.

ContentProviderT: Content providers encapsulate and manage data. They provide mechanisms, such as read and write permissions, to manage security. Architecturally, we distinguish read and write permissions on the data provided by these components.

BroadcastReceiverT: Broadcast receivers are components that receive system level events, like phone boot completed or battery low. We model broadcast receivers as distinct from activities because they can only receive a subset of intent types called standard broadcast action.

Figure 1 gives an example of an instance of the style showing two apps: K9-email (at the top) and PhotoStream. Each of the component types described above are represented by the rectangle or hockey puck shape with a solid line. We describe the connectors and how we represent apps below.

[6] http://developer.android.com/guide/components/services.html

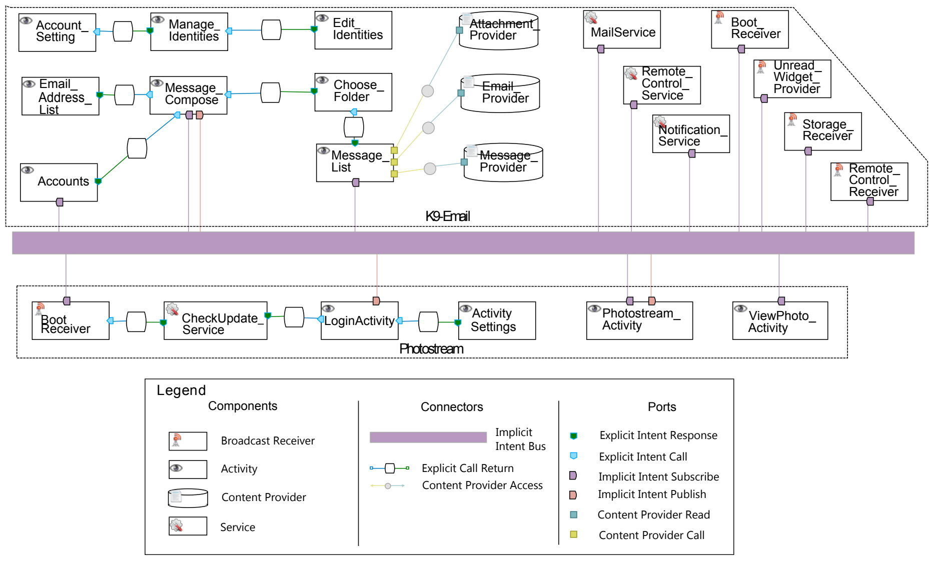

Fig. 1. An architecture instance in Android that captures two apps on a device.

Component type definitions for the style are relatively straightforward to derive, and are consistent with other work on modeling Android. However, port and connector modeling, in addition to modeling apps themselves, presents some challenges.

4.1 Modeling Apps as Groups

So far we have discussed how we have modeled elements of an app, but not the app itself. Because most vulnerabilities occur with inter-app communication, and apps themselves specify additional information (e.g., which activities are exported or public), we need a way to explicitly represent them. One way to do this would be via hierarchy: make each app a separate component with a subsystem that is composed of the activities, services, etc. This would mean that we could represent a device as a collection of App components, where the structure is hidden in the hierarchy. However, this complicates security checking. Because the checking needs understanding of sources and targets, and those sources and targets are activities and services (and not apps), any analysis would inevitably need to traverse the hierarchy.

Alternatively, Acme has a notion of *groups*, which are architectural elements that contain components and connectors as members. Like other architectural elements, they can define properties and rules. So, we use groups t to model apps. The group defines the permissions that an Android app has as a property. It then specifies its members as instances of the component types described above. Rules check that member elements do not require permissions that are not required by the app itself, providing some consistency checking. Groups naturally capture Android apps as collections of activities, services, content providers, etc., as well as the the case where communication can easily cross app boundaries by referring directly to activities that may be external to the app. Groups are shown in Figure 1 as dashed lines around the set of components that are provided by the app. Furthermore, for security analysis, groups form natural trust boundaries – communication within the app can be trusted because permissions are specified at the app level; communication outside the app should be analyzed because information flows to apps that may have different permissions. Therefore we also capture the permissions that are specified by apps as properties of the group. An application (group) specifies the set of permissions that an app is granted; activities specify the permissions that are required for them to be used.

4.2 Modeling Implicit and Explicit Intent Communication

One of the key requirements for enabling security analysis with formal models is being able to explicitly capture inter-app communication. All intents use the same underlying mechanism, but the *semantics* of implicit vs explicit intents are markedly different. Explicit intents require the caller to specify the target of the intent, and so are therefore more like peer-to-peer communication. Implicit intents require apps that can process the intent to specify their interest. Senders of the intent do not specify a receiver, and instead Android (or the user) selects which of the interested apps should process it through a process called intent resolution. This communication is more like publish subscribe. Because these different semantics suffer different vulnerabilities, they need to be separated in the style. In Figure 1 we can see one device-wide implicit intent bus as the filled in long rectangle in the middle of figure. Elements from all apps connect to this bus (the intent type and intent subscriptions are kept as properties on the ports of connected components).

Explicit intents are modeled as point-to-point connectors (pairwise rectangular connectors in Figure 1), where there is one source of the intent and one target. On the other hand, we model implicit intent communication via publish subscribe. We model one implicit intent bus per device. Implicit intents sent from components in all apps are connected to this bus; publishers specify the kind of intent that is being published (i.e., the intents action), whereas subscribers specify the intent filter being matched against. Different connector types for each intent-messaging type (i.e. explicit, implicit, and broadcast) allow for more nuanced and in-depth reasoning about security properties. For example, identifying unintended recipients of implicit intents is easier if implicit intents are first order connectors.

Android also has a notion of broadcasts (intents sent to broadcast receivers in apps). We did not define a separate connector for broadcasts because, for the purposes of security analysis, broadcast communication is done by sending intents (though via different APIs). Subscribing to broadcasts is also done by registering an intent filter, making both the sending and receiving for broadcasts the same as for intents.

5 Architecture Discovery

For security analysis to be tractable for an extendable system such as Android, we need to be able to derive the architecture from the system. Being able to do this means that a tool can be provided to construct Android architectures incrementally, and is the architecture is unique for each device. Figure 2 depicts the overview of our approach for recovering the architecture of an Android system. Given a set of Android application packages (also known as APKs), our architecture discovery method is able to recover the architecture of entire phone system. For this purpose, we leverage three components, namely, *Model Extractor*, *Template Engine*, and *Acme Studio*.

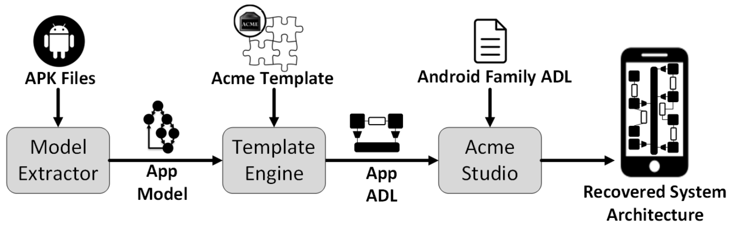

Fig. 2. Overview of Android system architecture discovery

The model extractor relies on the Soot [22] static analysis framework to capture an abstract model of each individual app. The captured model encodes high-level, static structure of each app, as well as possible intra- or inter-app communications. To obtain an app model, the model extractor first extracts information from the manifest, including an app's components, their types, permissions that the app requires, and permissions enforced by each component to interact with that component. It also extracts public interfaces exposed by each app, which are entry points defined in the manifest file through Intent Filters of components. Furthermore, the model extractor obtains complementary information latent in the application bytecode using static code analysis. This additional information, such as intent creation and transmission, or database queries, are necessary for further security analysis.

Once the generic model of an app (*App Model* in Figure 2) is obtained, the template engine translates it to an Acme-based architecture. Our template engine, which is based on FreeMaker library, needs a template (i.e., *Acme Template* in Figure 2) specifying the mapping between the app's extracted entities and the elements of Acme's architecture style for Android described in Section 4. The process of translating the app model (input) to an app architecture in the Acme ADL (output) is described in Algorithm 1.

Algorithm 1 consists of multiple iterations over three elements of apps (i.e., components, intents, and database queries) provided by the model extractor. It first iterates over the components of an app (lines 2–10), and generates a `component` element whose type corresponds to one of the four component types of Android (i.e., Activity, Service, Broadcast Receiver, and Content Provider). The properties of the generated components are set based on the extracted information from manifest (e.g., component name, permissions, etc.). If the type of a component is ContentProvider (line 5), a provider *port* is added to the component. Moreover, if a component has defined any public interface through IntentFilters (line 7), a receiver port is added and then, connected to the implicit Intents bus port, defined in the very beginning of the algorithm.

Afterwards, Algorithm 1 iterates over the communication messages, or Intents, of the given app model (lines 11–20). For the explicit Intents (line 12), two ports are added to the sender and receiver components of the Intent message, and then, a connector is generated to attach those ports. For implicit Intents (line 17), however, only one port

Algorithm 1: Translating Covert Model to Acme ADL

```
Input: $M$ : App model generated by COVERT
Output: $A$ : App ADL in accordance with Acme style for Android
1  $port_{bus} = A$.addConnector(ImplicitIntentBus)
2  foreach $comp_M \in M.components$ do
3      $comp = A$.addComponent($comp_M$)
4      $comp$.setProperties($comp_M$.getProperties())
5      if $comp_M.type$ = ''ContentProvider'' then
6          $comp$.addPort(ContentProviderResponsePort)
7      if $comp_M.IntetFilter \neq \emptyset$ then
8          $port_{receiver} = comp$.addPort(ImplicitIntentBroadcastReceivePort)
9          $conn = A$.addConnector(ImplicitIntentBroadcastReceive)
10         $conn$.attachPorts($port_{bus}, port_{receiver}$)
11 foreach $I_M \in M.Intents$ do
12     if $I_M.type$ = ''Explicit'' then
13         $port_{sender} = I_M.component$.addPort(ExplicitIntentCallPort)
14         $port_{receiver} = I_M.target$.addPort(ExplicitIntentResponsePort)
15         $conn = A$.addConnector(IntentCallResponseConnector)
16         $conn$.attachPorts($port_{sender}, port_{receiver}$)
17     else if $I_M.type$ = ''Implicit'' then
18         $port_{sender} = I_M.component$.addPort(ImplicitIntentBroadcastAnnouncerPort)
19         $conn = A$.addConnector(ImplicitIntentBroadcastAnnounce)
20         $conn$.attachPorts($port_{provider}, port_{bus}$)
21 foreach $Q_M \in M.DB\_Queries$ do
22     $port_{caller} = Q_M.component$.addPort(ContentProviderCallPort)
23     $port_{provider} = Q_M.authority$.getPort(ContentProviderResponsePort)
24     $conn = A$.addConnector(ContentProviderConnector)
25     $conn$.attachPorts($port_{caller}, port_{provider}$)
```

is added to the sender component of the message, which is attached to the bus port, already defined in line 1. Moreover, to capture data sharing communications, the algorithm iterates over database queries (lines 21–25), and adds a port to the components calling a ContentProvider. This port is then connected to the other port, previously defined (line 6) for the called ContentProvider, which is resolved based on the specified authority in the database query.

Finally, after translating app models of all APK files, generated ADLs are combined and together with Architecture description of Android framework (*Android Family ADL*) are fed into Acme Studio as the architecture of the entire system. This recovered architecture is further analyzed for identifying flaws and vulnerabilities that could lead to a security breach in the system.

6 Architecture Analysis of Android

We now describe how the Android-specific architectural models developed on top of Acme can be analyzed using both inherent analysis capabilities of Acme, as well as external analysis capabilities that require an architecture model of the system as input. To that end, we first describe two types of analyses supported innately by Acme: the ability to evaluate the conformance of architectural models to constraints imposed by the Android framework, and the ability to evaluate the architectural models against a predefined set of security threats. We then describe an integration of Acme with an external environment, called COVERT [5], that given an architectural representation of software is able to employ model-checking techniques to detect security vulnerabilities.

6.1 Conformance Analysis using Acme

In Section 4 we described the characteristics of the style, which are essentially derived from the constraints imposed by the Android framework on the structure and behavior of apps. Given the properties associated with permissions, exports, and intent filters, it is possible to describe well-formed architectures in this style via first order predicate logic rules. For example,

– Permission use within apps is consistent, meaning that any component of an app that has a permission must be declared also at the app level. This constraint is defined for each application group.

```
invariant forall m :! AppElement in self.MEMBERS |
  (hasValue(m.permission) −> contains (m.permission, usesPermissions));
```

– Explicit intent connectors should reference valid targets.

```
heuristic forall p in /self/components/ports:!ExplicitIntentCallPortT |
  exists c:!AppElement in self.components |
    c.class == p.componentReference;
```

– All implicit intents are attached to the global implicit intent bus.

```
invariant forall c1 :! IntentFilteringApplicationElementT in self.components |
  size (c1.intentFilters) > 0 −> connected (ImplicitIntentBus, c1);
```

– Activities and services that are not exported by an app are not connected to other apps.

```
invariant forall g1 :! AndroidApplicationGroupT in self.groups |
  forall g2 :! AndroidApplicationGroupT in self.groups |
    forall a1 :! IntentFilteringApplicationElementT in g1.members |
      forall a2 :! AppElement in g2.members |
        ((a1 != a2 and connected(a1, a2) and !a1.exported) −> g1 == g2);
```

Using these rules, Acme is able to check the architecture of individual apps, as well as a set of apps deployed together on an Android device. When used in a forward engineering setting, where a model of an app is constructed prior to its implementation, the analysis can find flaws early in the development cycle. When used in a reverse engineering setting, where a model of an app is recovered using the techniques described in Section 2, the rules can be applied to identify flaws latent in the implemented software.

6.2 Acme Security Analysis

Threats facing a system can be classified using STRIDE [21], which captures five different kinds of threat categories: Spoofing, Tampering, Repudiation, Denial of Service, and

Elevation of Privilege. According to the STRIDE model, a system faces security threats when it has information or computing elements that may be of value to a stakeholder. Such components or information are termed the *assets* of the system. Furthermore, most threats occur when there is a mismatch of trust between entities producing and those consuming the data. This approach conforms to the security level approach mismatch idea proposed in [12,13] and used by others since then (e.g., [7,14]).

STRIDE is often applied in the context of a larger threat modeling activity where the system is represented as a Data-Flow-Diagram. This representation is particularly useful for evaluating Android security issues that emerge from unintended intent passing. Viewing apps and the data they access as assets in terms of data flow exposes situations when possibly sensitive data passes between apps in an insecure way. For each data path between apps on a device, careful analysis can be performed to identify vulnerabilities, such as *spoofing* and *elevation of privilege* issues. Intent spoofing is a known classes of threat common in Android systems that occurs when a malicious activity is able to forge an intent to achieve an otherwise unexpected behavior. In one scenario the targeted app contains exported activities capable of receiving the spoofed intent. Once processed by the victim app can be leveraged to elevate the privileges of the malicious app by possibly providing access to protected resources.

Acme provides the framework for reasoning about app security. The properties needed to reason about these threats are present in terms of Android structures and data flow concerns. For example, Acme handles inter-app communication and exposes security properties about apps, such as whether they are exported and what permissions they possess. With this information in the model, automatically detecting app arrangements that may allow intent spoofing, information disclosure, and elevation of privilege can be written as first order predicate constraints over the style. Consider Listing 1 which shows how information disclosure vulnerabilities are detected. Each application group is assigned a trust level, based on the category of the app - for example, banking and finance apps would be more trusted than game apps; apps from certain providers like Google would have higher trust. The constraint specifies that if a source application sends an implicit intent to a target application then the source applications trust level must be lower than or equal to the recipient. These constraints for STRIDE are consistent with the approach taken in [15] for general data-flow architectures.

```
rule noInfoDisclosure = heuristic
    forall a1 :! ApplicationGroupT in self.GROUPS |
      forall a2 :! ApplicationGroupT in self.GROUPS |
        ((a1 != a2) −>
          (forall src :! ImplicitIntentBroadcastAnnouncerPortT in
              /a1/members:!ApplicationElementT/ports:!ImplicitIntentBroadcastAnnouncerPortT |
            forall activity :! ApplicationElementT in a2.members |
              forall tgt :! ImplicitIntentBroadcastReceiverPortT in activity.ports |
                (connected (src, tgt) and contains(src.action, tgt.intentFilters)) −>
                   a1.trustLevel <= a2.trustLevel));
```

Listing 1. Acme Constraint for Information Disclosure

This constraint (and others that are being checked) highlight potential pathways of concern and may generate false positives. This is one reason why in the style we specify the constraint as a heuristic (or warning), rather than as an invariant. These pathways would need to be more closely monitored at run time to determine whether the information should be transmitted.

6.3 Integrating with COVERT Security Analysis

In this section, we demonstrate how we can leverage the architectural models developed on top of Acme, together with external analysis environments that require such a model, to evaluate the security posture of an Android system. One such external environment employed in our research is COVERT [5], which provides the ability to automatically check inter-app vulnerabilities, i.e. whether it is safe for a combination of applications—holding certain permissions and potentially interacting with each other—to be installed simultaneously.

COVERT assumes that system architectural specifications are realized in a first-order relational logic [18]. Such specifications are amenable to fully automated yet bounded analysis. Specifically, the set of architectural models recovered by parsing individual apps installed on the device (cf. Section **??**) are first automatically transformed into the Alloy [18], a specification language based on relational logic, with an analysis engine that performs bounded verification of models.

The COVERT formal analyzer, in addition to extracted app specifications, relies on two other kinds of specifications: a formal architectural model of the Android framework and the axiomatized inter-app vulnerability signatures. Recall from Section 4, the architectural style for Android framework specification represents the foundation of Android apps. Our formalization of these concepts includes a set of rules to lay this foundation (e.g., application, component, messages, etc.), how they behave, and how they interact with each other. We regard vulnerability signatures as a set of assertions used to reify security vulnerabilities in Android, such as privilege escalation. All the specifications are uniformly captured in the Alloy language. As a concrete example, we illustrate the semantics of one of these vulnerabilities in the following. The others are evaluated similarly.

```
assert privilegeEscalation{
  no disj src, dst: Component, i:Intent|
    (src in i.sender) &&
    (dst in src.^transitiveIPC) &&
    (some p: dst.app.usesPermissions |
      not (p in src.app.usesPermissions) &&
      not ((p in dst.permissions) ||(p in dst.app.appPermissions)))
}
```

Listing 2. Specification of the privilegeEscalation assertion in Alloy.

Listing 2 presents an excerpt from an Alloy assertion that specifies the elements involved in and the semantics of the privilege escalation vulnerability. In essence, the assertion states that the victim component (dst) has access to a permission (usesPermission) that is missing in the src component (malicious), and that permission is not being enforced in the source code of the victim component, nor by the application embodying the victim component. As a consequence, an application with less permissions (a non-privileged caller) is not restricted to access components of a more privileged application (a privileged callee) [10].

Table 1. Performance of architecture extraction

Apps	Kilo # of Instructions	# of Components				# of Explicit Connectors	total # of Components & Connectors	Average Extraction Time (Sec)	
		Activity	Service	Receiver	Provider			Model	ADL
Amwell	1516	64	2	1	0	51	118	59.16	0.69
Audible's Audiobooks	2016	48	8	13	1	24	94	77.78	0.71
Baby Tracker	2163	79	30	46	4	90	249	84.76	0.73
BetterBatteryStats	388	9	3	6	0	23	41	18.43	0.66
Book Catalogue	119	21	0	0	1	21	43	31.93	0.65
K-9 Mail	835	28	7	5	3	38	81	33.48	0.63
LINE	2575	217	13	6	2	21	259	103.52	0.89
Mileage	128	50	2	2	1	18	73	9.54	0.62
MS Office Mobile	601	29	4	2	1	11	47	29.72	0.65
OctoDroid	447	53	0	0	0	31	84	21.88	0.64
Photo Grid	1771	54	5	3	1	32	95	73.42	0.74
SwiftKey	1159	35	13	19	0	16	83	49.28	0.68
Tango	1859	73	10	10	1	6	100	67.6	0.82
TextNow	1957	42	9	11	2	14	78	99.94	0.72
TouchPal	1538	88	6	16	0	81	191	66.19	0.78

The analysis is conducted by exhaustive enumeration over a bounded scope of model instances. Here, the exact scope of each element, such as Application and Component, required to instantiate each vulnerability type is automatically derived from the system architectural model. If an assertion does not hold, the analyzer reports it as a counterexample, along with the information helpful in locating the root cause of the violation. A counterexample is a certain model instance that makes the assertion false, and encompasses an exact scenario (states of all elements, such as components and Intents) leading to the violation.

7 Performance analysis

To evaluate the performance of our approach, we randomly selected and downloaded 15 popular Android apps of different categories from the Google Play repository, and ran the experiments on a computer with 2.2 GHz Intel Core i7 processor and 16 GB DDR3 RAM. We handled uncontrollable factors in our experiments by repeating them 33 times, the minimum number of repetitions needed to accurately measure the average execution time overhead at 95% confidence level. Table 7 summarizes the performance measurements for the architecture discovery process described in Section 5, divided into the time of model extraction and ADL generation.

Since the source code of the analyzed applications were not available, the (kilo) number of smali [7] instructions is used as the metric for the size of each app. Moreover, as an architectural metric, the number of components, categorized by the their types (i.e., Activity, Service, Broadcast Receiver, Content Provider), and explicit connectors, are provided in this table.

As shown in Table 7, there is a relationship between size (number of instructions) of the apps and model extraction time – apps with more instructions require more time to capture their model. On the other hand, the performance of the second part of the process, i.e., translating the extracted model to ADL, depends on the total number of components and connector, as the translator iterates over each of them.

[7] http://baksmali.com

8 Discussion and Future Work

In this paper, we have described an architectural style for Android that can be used to do various kinds of analysis to uncover, in particular, vulnerabilities related to inter-app intent communication. One of the challenges of doing such analysis in this domain is the dynamic nature of Android, and the need to understand all the related information flows. Statically forbidding certain kinds of information flows works against flexibility and prevents many valid, non-threatening communications from occurring. This paper provides a hybrid approach where the architecture (and information flows) of the system can be derived from analysis of the code and then can be used to analyze potential vulnerabilities on a per device basis.

The static analysis described in this paper identifies possible places where vulnerabilities may exist, but not actual exploits that may happen at run time. This requires combination of static analysis and run time analysis to capture and prevent actual exploits. Hence, static analysis can inform the run time analysis of parts of the system that need monitoring and deeper analysis, for example to examine the contents of intents, in order to determine if an exploit exists.

Our approach can be used to facilitate this combination of static and dynamic analysis. We are in the process of connecting our tool-suite to the Rainbow self-adaptive framework [16,8], where the vulnerabilities found statically can be used to choose adaptation strategies to change communication behavior in Android. We are in the process of addressing some of the challenges in integrating these two approaches, including disconnected operation and prevention of behaviors rather than reaction to behaviors.

For the modeling aspect, we have concentrated on understanding the architecture of the applications on the device, and those communication pathways. However, many apps are part of a large ecosystem with diverse back ends that are not on the device. Many of these apps may have information flows that affect security. How we model this, and how much, is an area of future work. Furthermore, security aspects are context-sensitive in the domain of mobile devices, where the degree of analysis required might change depending on whether devices are, for example, being used in a public coffee bar, or at home.

Acknowledgments

This work is supported in part by the National Security Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Security Agency or the U.S. government.

References

1. M. Abi-Antoun and J. M. Barnes. Analyzing security architectures. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 3–12, New York, NY, USA, 2010. ACM.
2. M. Almorsy, J. Grundy, and A. S. Ibrahim. Automated software architecture security risk analysis using formalized signatures. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 662–671, May 2013.

3. H. Bagheri, J. Garcia, A. Sadeghi, S. Malek, and N. Medvidovic. Software architectural principles in contemporary mobile software: from conception to practice. *Journal of Systems and Software (JSS)*. Under review.
4. H. Bagheri, E. Kang, S. Malek, and D. Jackson. Detection of design flaws in the android permission protocol through bounded verification. In *FM 2015: Formal Methods*, volume 9109 of *Lecture Notes in Computer Science*, pages 73–89. Springer International Publishing, 2015.
5. H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek. Covert: Compositional analysis of android inter-app permission leakage. *IEEE Transactions on Software Engineering*, 41(9):866–886, 2015.
6. H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek. Practical, formal synthesis and automatic enforcement of security policies for android. In *Proceedings of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016. To Appear.
7. C. Bodei, P. Degano, F. Nielson, and H. R. Nelson. Security analysis using flow logics. In *In Current Trends in Theoretical Computer Science*, pages 525–542. World Scientific, 2000.
8. S.-W. Cheng. *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation*. PhD thesis, Carnegie Mellon University, May 2008. Institute for Software Research Technical Report CMU-ISR-08-113.
9. E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *MobiSys '11*, 2011.
10. L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th international conference on Information security (ISC)*, 2010.
11. Y. Deng, J. Wang, J. J. P. Tsai, and K. Beznosov. An approach for modeling and analysis of security system architectures. *IEEE Trans. on Knowl. and Data Eng.*, 15(5):1099–1119, Sept. 2003.
12. D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.
13. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.
14. E. B. Fernandez, M. M. Larrondo-Petrie, T. Sorgente, and M. Vannhist. A methodology to develop secure systems using patterns. In *Integrating Security and Software Engineering: Advances and Future Visions*. Idea Group Inc., 2007.
15. K. Garg, D. Garlan, and B. Schmerl. Architecture based information flow analysis for software security, 2008.
16. D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10), October 2004.
17. D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*. 2000.
18. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, second edition, 2012.
19. J. Ren and R. Taylor. A secure software architecture description language. In *Workshop on Software Security Assurance Tools, Techniques, and Metrics*, pages 82–89, 2005.
20. A. Sadeghi, H. Bagheri, and S. Malek. Analysis of android inter-app security vulnerabilities using COVERT. In *ICSE 2015*, 2015. To Appear.
21. F. Swiderski and W. Snyder. *Threat Modeling*. Microsoft Press, Redmond, WA, USA, 2004.
22. R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.