Checking Intent-based Communication in Android with Intent Space Analysis

Yiming Jing†, Gail-Joon Ahn†, Adam Doupé†, and Jeong Hyun Yi‡
†Arizona State University ‡Soongsil University
{ymjing,gahn,doupe}@asu.edu, jhyi@ssu.ac.kr

ABSTRACT

Intent-based communication is an inter-application communication mechanism in Android. While its importance has been proven by plenty of security extensions that protect it with policy-driven mandatory access control, an overlooked problem is the verification of the security policies. Checking one security extension's policy is indeed complex. Furthermore, intent-based communication introduces even more complexities because it is mediated by multiple security extensions that respectively enforce their own incompatible, distributed, and dynamic policies.

This paper seeks a systematic approach to address the complexities involved in checking intent-based communication. To this end, we propose intent space analysis. Intent space analysis formulates the intent forwarding functionalities of security extensions as transformations on a geometric intent space. We further introduce a policy checking framework called IntentScope that proactively and automatically aggregates distributed policies into a holistic and verifiable view. We evaluate our approach against customized Android OSs and commodity Android devices. In addition, we further conduct experiments with four security extensions to demonstrate how our approach helps identify potential vulnerabilities in each extension.

1. INTRODUCTION

Modern mobile operating systems have shifted into a security architecture that is fundamentally different from those of traditional desktop OSs. Mobile applications (commonly referred to as apps) run as unique security principles; they are isolated in their respective sandboxes and receive few privileges. Despite that apps are isolated, they interoperate through inter-application communication. As such, a few apps, whose workflows are directed by a user, can accomplish complex and diversified tasks. For example, an email client exports a picture file to a photo editor; the photo editor modifies the picture and posts it online through a social network client.

ASIA CCS '16, May 30-June 03, 2016, Xi'an, China
 ISBN 978-1-4503-4233-9/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2897845.2897904

A type of messaging objects called *intents* build a major and sophisticated inter-application communication mechanism in Android [13]. Intents are flexible as they can carry simple data and even inter-process communication primitives (*e.g.* Binder [2] and file descriptors [3]). Moreover, the intent attributes are rich with Android middleware semantics, which naturally facilitate access control decisions [12, 29]. As a result, the security community proposed plenty of security extensions that implement policy-driven mandatory access control (MAC) for intent-based communication [7,10–12,21,26,27,29,32,38]. Indeed, intents are not the only inter-application mechanism in Android. The recent MAC implementations [7,12,32] adopt derivations of SELinux kernel MAC to cover the other Android mechanisms such as files, sockets, and Binder. However, intents are out of the scope of kernel MAC due to the incompatible semantics of the kernel and middleware layers [12].

Defining and verifying the policy for each individual security extension that controls intent-based communication is a complex task for a policy analyst. The recent emerging security requirements, such as "bring your own device" (BYOD), call for fine-grained and precise policies. For example, a single mobile device may host a doctor's personal apps and the apps of several clinics. The doctor and the clinics would require that the deployed security policies accurately enforce the boundaries between the apps of the respective stakeholders. Meanwhile, mitigating existing threats related to intents such as communication hijacking [13], confused deputy attacks [10, 19], and accidental data disclosure [26] requires that policies are tailored to the peculiarities of each threat and each vulnerable app.

Furthermore, the complexity significantly increases when intent-based communication is mediated by multiple collaborating security extensions that enforce their respective security polices. First, the security extensions define *incompatible* schemes, logic, and semantics for their policies. Second, the policies that determine how intents are processed and forwarded among apps are *distributed* across multiple security extensions. Moreover, the policies are stored and updated in a *dynamic* manner due to frequent app installs, uninstalls, and upgrades. As a consequence, a policy analyst must manually inspect every security extension's policy, aggregate them into a holistic view, and search for violation of security properties. Overall, policy verification becomes an error-prone and tedious task that requires great sophistication from the policy analyst. This leads to slow adoption of Android security extensions despite that quite a few modern security extensions have been proposed recently.

To effectively address the complexities of checking intent-based communication, we argue for the need of a general, holistic, and proactive policy checking framework that analyzes the incompatible, distributed, and dynamic intent-based communication policies in Android. In particular, the framework automatically aggregates the policies and generates a holistic view that lends itself well for formal verification. The tools that currently exist are dependent and specialized to each security extension. For example, EASE-Android [33] and SETools [4] are tailored to SEAndroid and SELinux. To the best of our knowledge, we propose the first tool to holistically check multiple security extensions.

This paper proposes *intent space analysis* to address the complexities in checking intent-based communication. Intent space analysis is built upon a geometric *intent space model*. In this model, we propose to represent intents with a K-dimensional space of regular languages, in which each dimension corresponds to an intent attribute. As such, an intent maps to a point in the space, multiple intents map to a subspace, and security extensions are modeled as transfer functions that map one subspace to another. For example, a security extension that denies any intent can be modeled as a transfer function that maps all K attribute values (denoted as $\{.*\}^K$) to an empty space regardless of source or destination apps.

We further propose a policy checking framework, called IntentScope. Given an Android device, IntentScope acquires and parses the live intent forwarding states of each security extension that controls intents. Afterwards, IntentScope automatically instantiates transfer functions from the acquired states. By composing the transfer functions, IntentScope constructs a *snapshot* of the holistic intent forwarding state as a graph whose vertices correspond to apps and whose edges correspond to system-allowed intents (as intent spaces) between apps. The graph supports flexible queries and facilitates novel security assessment tasks such as checking domain isolation, enumerating UI workflows [26], and discovering permission re-delegation paths [19].

This paper makes the following main contributions:

- We propose an intent space model for modeling intent-based inter-application communication in Android. Our intent space model is general and independent of specific security extensions.
- We propose IntentScope, a general, holistic, and proactive policy checking framework for intent-based communication. IntentScope reasons about a holistic graph derived from the live intent forwarding states maintained by multiple security extensions in an Android device.
- We implement a prototype of IntentScope and evaluate it against mainstream security extensions, commodity Android devices, and customized Android OSs. We also showcase a series of novel analysis tasks that help a policy analyst discover weak points in policies.

The remainder of this paper proceeds as follows. Section 2 provides the background and problem description of our work. Section 3 describes our intent space model. Section 4 introduces IntentScope and describes its system design followed by experimental results in Section 5. Section 6 discusses limitations and future work. Section 7 overviews related work. Section 8 concludes this paper.

2. BACKGROUND

In this section, we first discuss the background of intent-based communication in Android. We then present the problem description of this work.

2.1 Intent-based Communication

Components are the basic building blocks of Android apps. There are four types of components, and each type serves a specific purpose:

- **Activities:** An activity represents the user interface.
- **Services:** A service has no user interface and runs in the background for time-consuming operations.
- **Content providers:** A content provider exposes an app's data as tables and supports basic operations such as insert, delete, and update.
- **Broadcast receivers:** A broadcast receiver is triggered upon system or application events.

A component can be exported to other apps. Each exported component of an app is an entry point for intents through which the other apps or the Android system can send intents. Typically, an app exports its components to other apps by statically declaring the exports in the app's manifest[1]. However, an app can also dynamically create and export components in its code. Two system services, PackageManagerService (PMS) and ActivityManagerService (AMS), maintain the information about each installed app's components regardless of how the components are exported—either statically or dynamically.

Intents can connect an app's component to exported components. An app creates an intent and sets its embedded attributes. The intent is then processed by the Android system and the security extensions, which automatically resolve an intent's recipients based on the following intent attributes:

- **Component name:** This attribute explicitly specifies the expected recipient of the intent.
- **Action:** This attribute describes the general action to be taken by a recipient component, such as `PICK`, `VIEW`, `EDIT`, or `SHARE`.
- **Scheme:** This attribute describes the protocol that serves the data, such as `http`, `mailto`, or `tel`.
- **Authority:** This attribute describes the location of the data, such as `www.google.com` or `paypal`.
- **Type:** This attribute describes the MIME type of the data, such as `audio/ogg`, `video/*`, or `*/*`. Note that wildcards are allowed.
- **Category:** This attribute provides additional information about the data. For example, a category `BROWSABLE` implies the data that can be opened in a web browser, such as a link to an image.

Two types of intents exist in Android. *Explicit intents* specify the component name only. Android delivers an explicit intent directly to its specified component regardless of the presence of any other attributes. *Implicit intents* specify the attributes other than component name. Thus, an implicit intent's recipients are implicit and must be resolved at intent-sending time; Android must search the registered components to resolve the recipient components.

[1]The manifest is a required XML file included in the app by the developer.

2.2 Problem Description

Intent-based inter-application communication has received much research attention. In general, two aspects are covered: previously unknown security limitations of intents [10, 13, 15, 19, 25] and generic policy-driven security extensions that remedy the limitations [7, 10–12, 21, 26, 27, 29, 32, 38]. However, there is an overlooked gap between configuring generic security extensions and securing a specific Android device. Every app, every device, and every user are different. A policy analyst needs insights about intent-based communication before she can accurately define how the apps in her device communicate through intents in her intended ways. To bridge the gap, we seek a systematic approach for a policy analyst to conveniently acquire such insights.

Intent-based communication is mediated by multiple security extensions. While multiple security extensions promote the flexibility of controlling intent-based communication, they also introduce new challenges in definition and verification of their policies.

C-1: *Incompatible policies.* The security extensions define their own incompatible schemas and semantics. For example, FlaskDroid [12] inherits SELinux's policy semantics of type enforcement. Saint [29] uses an XACML-like schema customized by the authors. IntentFirewall's policy is unique and unlike the other security extensions, however it specifies a critical set of tests on intent attributes. As far as we know, no existing policy checker can work with every extension's policy. Therefore, checking such incompatible policies remain a manual process that requires a policy analyst to master the details of every security extension.

C-2: *Local policies.* Each security extension manages a local view of system-wide policies. For example, IntentFirewall enforces its centralized policy specified by a policy administrator; intent filters manages policies specified by decentralized apps but enforce the policy in a centralized manner. Each security extension makes its decision by itself and is not aware of the other security extensions. No security extension possesses a holistic view of the reachability among installed apps as controlled by all the security extensions.

Problem Statement. To address the challenges in checking intent-base communication, we seek to build: *a*) a *general* policy checker that easily adapts to the policy schema of any security extension that controls intents; *b*) a *proactive* policy checker that keeps monitoring the live intent forwarding states of security extensions; and *c*) a *holistic* policy checker that aggregates the policies into a holistic and verifiable view. With the policy checker, we attempt to systematically answer the following two questions regardless of specific security extensions, apps, or devices: *a*) what intents can an app send to a specific app; and *b*) what intents can an app receive from a specific app. Meanwhile, we expect the checker to be mostly automated so as to reduce the burden on policy analysts.

Assumptions. In this work, we assume an Android device is loaded with multiple policy-driven security extensions that mediate intent-based communication among installed apps. The apps could be malicious, but they do not seek to escape from the confinement of the security extensions. In other words, the policies define apps' capabilities to send or receive intents. Threats that compromise the integrity of the Android system, the security extensions, and the policies are beyond the scope of our approach.

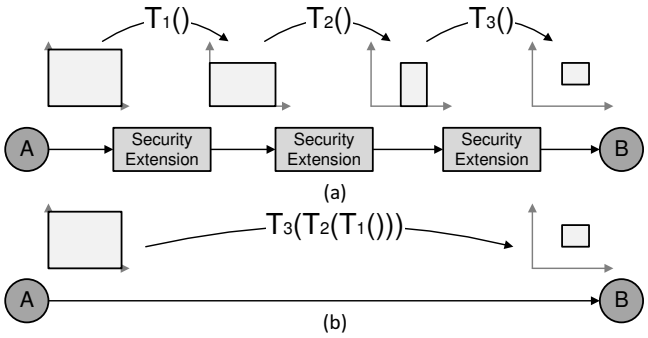


Figure 1: (a) The intent space from App A to App B shrinks as it passes security extensions, modeled here by the T_1, T_2, T_3. (b) Composing transfer functions to model app-to-app transformation.

3. INTENT SPACE ANALYSIS: MODEL

We believe that creating the right abstraction model is the first step toward checking intent-based communication. In this section, we elaborate the intent space model that lays the foundation for intent space analysis.

Figure 1 demonstrates a motivating example where App A sends intents to App B. For simplicity of the example, we consider only actions and categories, and we represent the actions on the x-axis and the categories on the y-axis. The initial space of App A is full in both dimensions because an app can create arbitrary intents before the intents are processed by any security extension. And because the security extensions only forward the intents that match certain actions or categories specified in their policies, the space gradually shrinks as the transformations T_1, T_2, and T_3 are applied to the initial space (Figure 1 (a)). The remaining space at App B indicates the intents that App A can send to reach App B. And if no space remains, App A cannot communicate with App B through intents. One step further, we combine the transfer functions into a composite transfer function that describes app-to-app space transformation as illustrated in Figure 1 (b). This composite function captures all the security extensions. Thus, it describes the holistic intent forwarding state.

3.1 Intent Space

Formally, an intent space is a K-dimensional space of regular languages defined as $\mathcal{I} = \{.*\}^K$, where "`.*`" is the regular language that describes all words. The K dimensions correspond to K intent attributes, which are selected by the policy analyst based on her requirements. A policy analyst can set a smaller K if the security extensions to be analyzed do not inspect every intent attribute. An intent i maps to a point in the space, such as: {`action`: `SEND`,`category`: `DEFAULT`}[2] for $K = 2$. Multiple intents map to a subspace defined as a hypercube or a union of multiple hypercubes. A hypercube is represented with *exactly* K regular languages at K dimensions, such as {`action`: `SEND|SEND_MULTI`, `category`: ε (the empty string language)}. Any hypercube with fewer than K dimensions or undefined dimensions is invalid and considered as an empty space $\varnothing$ in the subsequent computations.

[2]For clarity in this example we annotate the dimensions with the attributes.

3.2 Intent Space Algebra

Algorithms that check intent-based communication between two apps must determine whether an app's allowed outgoing intents overlap with the other apps' allowed incoming intents. To this end, we define the basic set operations on $\mathcal{I}$: *intersection*, *union*, *complementation*, and *difference*. Note that a point in $\mathcal{I}$ can be considered as a special hypercube whose regular languages contain only one word; and a subspace is a union of multiple hypercubes. We therefore define set operations for hypercubes and carry over the operations to other intent space objects. Throughout the rest of this paper, we overload the term *intent space* to refer to all types of intent space objects including points, hypercubes, subspaces, as well as the entire intent space.

Intersection. The intersection of two intent spaces is computed by intersecting the regular languages at each dimension. Formally, given two intent spaces $i, j \subset \mathcal{I}$ and their dimension set $D = \{d_1, d_2, \ldots, d_k\}$, their intersection $i \cap j$ is $\{d_1 : regex_1^i \cap regex_1^j, \ldots, d_k : regex_k^i \cap regex_k^j\}$. For example, {`A[12]`, `C1`} $\cap$ {ε, `C1`} is equivalent to {`A[12]`, `C1`} and {`A[12]`, `C1`} $\cap$ {`A3`, `C1`} is equivalent to {$\varnothing$, `C1`}. Note that {$\varnothing$, `C1`} is missing a dimension and thus is considered as an empty space $\varnothing$.

Union. A union of intent spaces may not be simplified to a single intent space. For example, the union of two intent spaces {`A1|A2`, `C1`} and {`A3`, `.*`} cannot be represented by any single hypercube and we simply represent the union as {`A1|A2`, `C1`} $\cup$ {`A3`, `.*`}. We can simplify the result if the intent spaces are on the same hyperplane. For example, {`A1|A2`, `C1`} $\cup$ {`A3`, `C1`} is equivalent to {`A[1-3]`, `C1`}.

Complementation. The complement of an intent space i is the union of all the intent spaces that do *not* intersect with i. Recall that the intersection of two intent spaces is an empty space if the intersection is missing any of the K dimensions. We compute i's complement $\overline{i}$ with Algorithm 1, which finds all non-intersecting intent spaces by replacing the regular language at one dimension with its complement if the language is not `.*` and setting `.*` at the other dimensions. For example, the complement of {ε} is {.*} and the complement of {`A1`, `C1`} is {$\overline{\texttt{A1}}$, `.*`} $\cup$ {`.*`, $\overline{\texttt{C1}}$}.

Algorithm 1: Computing an intent space's complement

```
Data: i
Result: ī
i' ← ∅;
for dimension d_i ∈ D do
    L ← regular language at d_i;
    if L ≠ .* then
        i' ← i' ∪ {d_1 : .*, ..., d_i : L̄, ..., d_k : .*};
return i'
```

Difference: The difference (or subtraction) is computed with intersection and complementation, *i.e.*, $i - j = i \cap \overline{j}$. For example, {`A1|A2`, `.*`} - {`A2`, `.*`} is equivalent to {`A1|A2`, `.*`} $\cap$ {$\overline{\texttt{A2}}$, `.*`}, which is {`A1`, `.*`}. A slightly more complicated example which reuses the complement of {`A1`, `C1`} is:

$$\begin{aligned}
&\{\texttt{A1|A2}, \texttt{C1|C2}\} - \{\texttt{A1}, \texttt{C1}\} \\
&= \{\texttt{A1|A2}, \texttt{C1|C2}\} \cap \overline{\{\texttt{A1}, \texttt{C1}\}} \\
&= \{\texttt{A1|A2}, \texttt{C1|C2}\} \cap (\{\overline{\texttt{A1}}, \texttt{.*}\} \cup \{\texttt{.*}, \overline{\texttt{C1}}\}) \\
&= \{\texttt{A2}, \texttt{C1|C2}\} \cup \{\texttt{A1|A2}, \texttt{C2}\}
\end{aligned}$$

3.3 Transfer Function

For convenience of analysis, we assume that all security extensions deny by default. For those security extensions that accept by default, it is trivial to reduce them into deny-by-default extensions with a least-priority rule that accepts everything. Therefore, apps cannot communicate if the security extensions specify no rule. Conversely, the rules of a security extension that allow/deny some intents from one app to another app essentially specify how the security extension *forwards* or *drops* intents from the source app to the destination app. As we represent intents as an intent space, we model a security extension's intent forwarding and dropping functionality as intent space transformation and represent a security extension with a *transfer function*. Given that the space of all apps is $\mathcal{A}$, a transfer function T is:

$$T : (a, i) \rightarrow 2^{\mathcal{A} \times \mathcal{I}}, a \in \mathcal{A}, i \subset \mathcal{I}$$

To aggregate multiple transfer functions into a holistic view, we iteratively apply each (a, i) tuple of the output of a transfer function to the input of the next transfer function and build a composite transfer function.

A transfer function captures the transformation that a security extension performs on $\mathcal{A}$, $\mathcal{I}$, or both. Suppose we are to model a simple security extension that works like a Layer-2 network switch: it only supports coarse-grained control over which app can send intents to another app regardless of intent attributes. Such an extension can be modeled as a transfer function that transforms only on $\mathcal{A}$. IntentFirewall denies an app from sending a specific intent regardless of the intent's destination apps. It therefore can be modeled as a transfer function that only transforms on $\mathcal{I}$. We elaborate more details about how we model security extensions for intent space analysis in the subsequent section.

4. INTENT SPACE ANALYSIS: SYSTEM

In this section, we describe our policy analysis framework IntentScope which supports intent space analysis. To demonstrate its generality, we also discuss how IntentScope works with the security extensions for intents in Android Open Source Project (AOSP) and their policies. We emphasize that IntentScope is not limited to only the discussed security extensions in this paper.

Figure 2 depicts the workflow of IntentScope. In general, IntentScope starts from acquiring the policies of security extensions, then creates transfer functions, and converts the composite transfer function into a holistic reachability graph for subsequent analysis.

Acquiring Policies. The policy of a security extension is often referred to as a dedicated file stored in the filesystem. In this work, we opt for a more general definition of policy and propose to acquire all the states and configurations of security extensions so long as they specify how the intents are forwarded. To this end, we create a privileged watchdog app for IntentScope that proactively observes policy changes and automatically takes a snapshot of the policies. The implementation of the watchdog app is largely specific to the analyzed security extensions. For example, intent filters are registered by apps and maintained by AMS and PMS. The watchdog app acquires the registered intents filters on an Android device by dumping the internal states of AMS and PMS after an app registers/unregisters any intent filter.

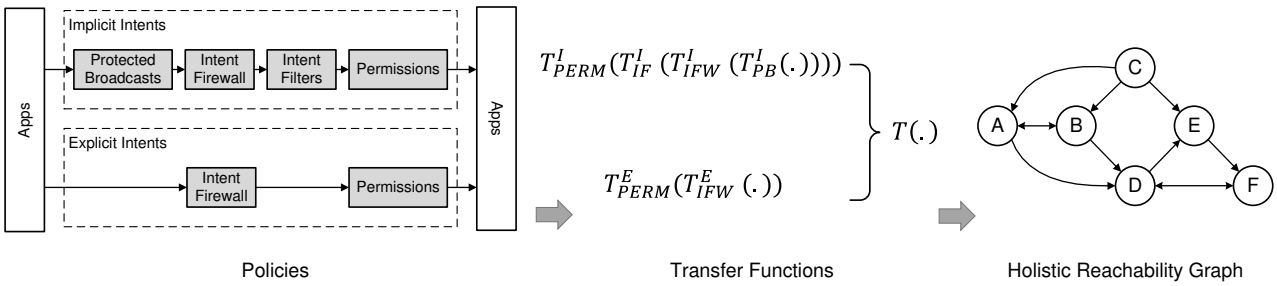


Figure 2: IntentScope System Workflow.

Creating Transfer Functions. Next, we map the acquired policies onto transfer functions. Given that a security extension makes decisions based on its loaded *policy* and implemented policy interpretation *logic*, a transfer function that models the intent forwarding state must capture both. While the policy can be automatically retrieved by IntentScope's watchdog app, the policy interpretation logic still requires manual effort to model. IntentScope requires the security extension's authors or policy analysts to define a transfer function for its policy interpretation logic and to create a policy parser that instantiates the corresponding transfer function. Note that this logic construction overhead is only performed once as the defined transfer functions can be reused and the parsers can automatically instantiate transfer functions. We elaborate our transfer functions for the AOSP security extensions in Section 4.1.

Building a Holistic Reachability Graph. To facilitate analysis and visualization, we propose to convert the composite transfer function into a directed graph that represents inter-application reachability. Formally, a holistic reachability graph is denoted as $G = (V, E)$, where V is a set of vertices that correspond to the installed apps and E is a set of edges that correspond to the intent spaces that an app can send to reach another app. Constructing such a reachability graph is straightforward. Each app maps to a vertex in the graph. For each app, we apply the composite transfer function on its initial intent space (*e.g.*, $\{.*\}^K$) and add a directed edge if any non-empty intent space remains at the destination app. We assign the remaining intent spaces on the edges as their weights, which allows IntentScope to support flexible queries and graph pruning as a policy analyst adds constraints on the graph.

4.1 Modeling AOSP Security Extensions

Intent filters, IntentFirewall, protected broadcasts, and permissions are the integral parts of AOSP and therefore widely deployed in COTS Android devices. They also serve as reference implementations for other security extensions. For example, Apex [27] and CRePe [14] extend the permissions; and SEAndroid controls intents with a slightly modified IntentFirewall [5]. Based on these observations, we believe that the AOSP security extensions are a good starting point to demonstrate that IntentScope is general, because it can effectively work with their policies. In the remainder of this section, we share our experiences of modeling these security extensions for intent space analysis. Although we are not the first to formally model them, we provide the most accurate models by covering a complete set of intent attributes and undocumented logic in the security extensions. Unless stated otherwise, the contents in this section are based on the `kitkat-release` branch in AOSP.

As shown on the left side of Figure 2, two chains of security extensions control implicit and explicit intents. We define two intent spaces: (1) $\mathcal{I}_I$ as a six-dimensional implicit intent space over five intent attributes `action`, `category`, `scheme`, `authority`, `type` and one additional attribute `permission`; and (2) $\mathcal{I}_E$ as a two-dimensional explicit intent space over `component name` and `permission`. Note that the `permission` of an intent is inherited from the app that created the intent. The chain for implicit intents consists of four security extensions: protected broadcasts, IntentFirewall, intent filters, and permissions; and we define their transfer functions over $\mathcal{I}_I$ as T^I_{PB}, T^I_{IFW}, T^I_{IF}, and T^I_{PERM}. The chain for explicit intents includes two security extensions: IntentFirewall and permissions; and we define their transfer functions over $\mathcal{I}_E$ as T^E_{IFW} and T^E_{PERM}.

4.1.1 *Intent Filters:* T^I_{IF}

An intent filter specifies the implicit intents that it allows to be forwarded to the next security extension. Therefore, an intent filter's output is the intersection of the input intent space and the intent filter's corresponding intent space. Suppose a component *dst.c* in an app *dst* has an intent filter *filter* that describes an intent space $i^{dst.c}_{filter}$. Then, an intent filter transforms (src, i) to $(dst, i \cap i^{dst.c}_{filter})$. Note that the transformation is performed on both $\mathcal{A}$ and $\mathcal{I}$. Given the installed apps on a device as a set A, we combine their registered intent filters and define T_{IF} as follows:

$$T^I_{IF}(m, i) = \{(n, i \cap i^{n.c}_{filter}) | i \cap i^{n.c}_{filter} \neq \varnothing,$$
$$\forall c \text{ is a component of } n, \forall n \in A, n \neq m; i, i^{n.c}_{filter} \subset \mathcal{I}_I\}$$

Next we explain how we map an intent filter to its intent space i_{filter}. In general, an intent filter accepts an intent if the intent's attributes pass a series of tests on the intent filter's attributes. Therefore, we reduce the problem of modeling an intent filter to constructing a set of regular languages which consists of the words that pass each test.

Action Test: An intent passes the action test if the intent's action matches any action in the intent filter. Therefore, we map the one or more actions of an intent filter onto a regular expression that concatenates the *escaped* action strings and separates them with the vertical bar character `|`, such as `VIEW|EDIT`. There are two corner cases in this test. First, zero action in a filter fails the test. Second, zero action in an implicit intent also fails the test. We capture both cases with a regular expression `[]`, which denotes an empty language whose intersection with any language is empty. Note that the Android documentation is incorrect with respect to the second corner case: "*if an intent does not specify an action, it will pass the test as long as the filter contains at least one action*". The reason is that `queryIntent()`

in the `IntentResolver` class eventually denies such intents even though `matchAction()` in the `IntentFilter` class allows. Our experiments also confirm this behavior. Interested readers are referred to the source code[3].

Scheme Test: An intent passes the scheme test if the intent's scheme matches any scheme in the filter. Therefore, the regular expression here is constructed in the same way as the action test, *e.g.*, `http|gopher`. This test also has unique cases. First, an intent filter without any scheme still matches three schemes: `content`, `file`, or an empty string. We represent them with a regular expression `file|content|`, where the last `|` matches the empty string. Second, an intent without any scheme passes the scheme test only if the intent filter does not specify any scheme. We consider such intents as intent spaces whose scheme is an empty string.

Authority Test: This test is dependent on the scheme test. If the intent filter does not specify any scheme, this test automatically passes regardless of the authority. This test also passes if the filter does not specify any authority. Thus, we use `.*` to match any authority in these two cases. An intent without any authority passes the test only if the filter has no authority. We represent such intents with an empty string at the authority dimension. Otherwise, an intent passes the authority test if its authority matches any authority in the filter.

Type Test: An intent passes the type test if the intent's MIME type matches any type in the filter. The challenge here is the wildcard character `*` in MIME type strings. For example, `*` and `*/*` match any type; and `audio/*` matches any subtype of `audio`. To maintain the semantics of the wildcard character, we convert `*` and `*/*` to `.*`. The slash character `/` is a special character in regular expressions so we escape it as `\/`. For example, `audio\/.*|video\/mp4` represents every audio subtype and a single video type. Moreover, an intent filter that has no type accepts only the intents that have no type. Therefore, zero type in either the intent or the filter maps to an empty string.

Category Test: Unlike the other attributes, an intent can include more than one category. An intent passes the category test if *every* category in the intent matches a category in the filter, *i.e.*, the intent's category set is the subset of the filter's category set. To capture this logic, we construct a regular language for an intent filter's categories with three steps: (1) escape the category strings; (2) concatenate the escaped strings and separate them with `|`; and (3) surround the concatenated string with `(` and `)*`. For example, the subsets of an intent filter's category set {`DEFAULT`, `LAUNCHER`, `BROWSABLE`} are represented with a single regular expression `(DEFAULT|LAUNCHER|BROWSABLE)*`. This expression also matches zero category and duplicate categories specified in an intent. The other corner cases are similar to those of the type test. No specification of category in an intent or a filter maps to an empty string. An intent filter with no category accepts only the intents with no category.

4.1.2 *IntentFirewall:* T^I_{IFW} *and* T^E_{IFW}

IntentFirewall is a policy-driven MAC framework that block apps from *sending* specific intents. The policy files, located at `/data/system/ifw/*.xml`, specify a list of *firewall filters* (fwfilters for short) that describe the implicit or explicit intents to be blocked for a specific sender app. We model IntentFirewall as a transformation over $\mathcal{I}_I$ or $\mathcal{I}_E$ that subtracts the intent space of each fwfilter from the input intent space. Suppose a fwfilter that blocks an app *src* is represented with an intent space $i^{src}_{fwfilter}$. T^I_{IFW} and T^E_{IFW} are defined in the same way as follows:

$$T^I_{IFW}(a,i) = \{(a, i - \bigcup i^a_{fwfilter}) | i - \bigcup i^a_{fwfilter} \neq \varnothing,$$
$$\forall fwfilter \text{ that blocks the sender app } a; i, i^a_{fwfilter} \subset \mathcal{I}_I\}$$
$$T^E_{IFW}(a,i) = \{(a, i - \bigcup i^a_{fwfilter}) | i - \bigcup i^a_{fwfilter} \neq \varnothing,$$
$$\forall fwfilter \text{ that blocks the sender app } a; i, i^a_{fwfilter} \subset \mathcal{I}_E\}$$

Next we explain how we construct the intent space $i_{fwfilter}$ for a fwfilter over the implicit intent space $\mathcal{I}_I$ and the explicit intent space $\mathcal{I}_E$, respectively. In general, we construct $i_{fwfilter}$ according to IntentFirewall's two-phase intent attribute matching process.

If a fwfilter is for implicit intents, IntentFirewall first considers the fwfilter as an intent filter and tests the intent attributes with the same tests as we discussed in Section 4.1.1. We skip modeling this phase for brevity. In the second phase, IntentFirewall tests the intent attributes with common string tests, such as `isEqual`, `isStartsWith`, `isContained`, and `matchRegex`. Therefore, we model these tests with their equivalent regular expressions. For example, `isStartsWith=abc` maps to a regular expression `abc.*`; `isContained=def` maps to a regular expression `.*def.*`. The tests can be aggregated by computing the intersection of the regular expressions. For example, two tests `isEqual=abc` and `isStartsWith=ab` map to a regular expression `abc`.

For a fwfilter that filters explicit intents, we also construct its intent space in two phases. In the first phase, IntentFirewall checks if an explicit intent's component name matches the one specified in the fwfilter. Thus, we simply copy the fwfilter's escaped component name to the corresponding dimension in $i_{fwfilter}$. There are two corner cases to be handled. An explicit intent with no component name is dropped immediately because it resolves to nowhere. A fwfilter with no component name does not block any explicit intent. We model the former case with a regular expression `[]` and model the latter case with a regular expression `.*`. In the second phase, Intent Firewall tests the intent's component name with the identical string tests so we do not rephrase how we model them. Finally, both T^I_{IFW} and T^E_{IFW} do not transform an intent space at the permission dimension because IntentFirewall does not inspect permissions.

Note that IntentFirewall is a relatively new security extension in AOSP with no official documentation and limited comments in the code. At first we referred to the unofficial documentation maintained by Yagemann [36] to define the transfer functions. However, we found unexplained behaviors of IntentFirewall when we tested IntentFirewall's sample policies, which led us to the discovery of the overlooked second matching phase. In order to obtain an accurate and comprehensive model, we manually derived the transfer functions presented in this section from IntentFirewall's source code[4].

4.1.3 *Permissions:* T^I_{PERM} *and* T^E_{PERM}

Permissions constrain an app's capability to *receive* intents from other apps. Suppose an app has a sensitive component that only accepts the intents from authorized apps. Then, the app can define a permission and assign it

[3]https://goo.gl/A1auU5 and https://goo.gl/cdzxg8

[4]https://goo.gl/e4zzxL

to the component, which requires the component's callers to hold the exact same permission. If we treat intents as if they inherit the permissions of their creator/sender apps, a permission's role is to forward only the intents that have matching permissions. Therefore, a permission's output is the intersection of the input intent space and the permission's own intent space. Note that permissions do not transform on $\mathcal{A}$ because the other security extensions have already resolved the destination app/component. Suppose a component $dst.c$ is protected by a permission p described by an intent space $i_{c.p}^{dst.c}$. The transformation is defined as $(dst.c, i) \rightarrow (dst.c, i \cap i_{c.p}^{dst.c})$.
We define T_{PERM}^{I} and T_{PERM}^{E} as follows:

$$T_{PERM}^{I}(a,i) = \{(a.c, i \cap i_{c.p}^{a.c}) | i \cap i_{c.p}^{a.c} \neq \varnothing, \\ \forall c \text{ is a component of } a; \\ c \text{ is protected by } c.p; i, i_{c.p}^{a.c} \subset \mathcal{I}_I\}$$

$$T_{PERM}^{E}(a,i) = \{(a.c, i \cap i_{c.p}^{a.c}) | i \cap i_{c.p}^{a.c} \neq \varnothing, \\ \forall c \text{ is a component of } a; \\ c \text{ is protected by } c.p; i, i_{c.p}^{a.c} \subset \mathcal{I}_E\}$$

Mapping a permission to an intent space i_p is straightforward. The regular language at the permission dimension of i_p is the escaped permission string. A special case is that a content provider may have separate permissions for reading and writing. Similar to the action test in intent filters, we model this case with a regular expression `perm_r|perm_w`, based on the fact that an app with either the read or write permission can access the content provider. The regular languages at the other dimensions are `.*`, leaving the intent space unchanged at these dimensions.

4.1.4 Protected Broadcasts: T_{PB}^{I}

Protected broadcasts are a set of implicit intents with special actions that only the apps whose UIDs are `SYSTEM`, `BLUETOOTH`, `PHONE`, or `SHELL` can send. The other apps are prevented from sending such intents. Similar to IntentFirewall, we model protected broadcasts as a space transformation that subtracts the intent spaces of protected broadcasts from the input intent space if the input app is not a system-app. Suppose each protected broadcast maps to an intent space $i_{protected}$. Then, we define the transfer function for protected broadcasts as follows:

$$T_{PB}^{I}(a,i) = \begin{cases} (a,i) & \text{if } a \text{ is an allowed app} \\ (a, i - \bigcup i_{protected}) & \text{otherwise} \end{cases}$$
$$i, i_{protected} \subset \mathcal{I}_I$$

A list of actions used by protected broadcasts is available in the Android SDK[5]. Thus, we build an intent space $i_{protected}$ for each action by assigning the escaped action string into the action dimension of the space. The other dimensions do not involve space transformation and remain with a regular expression `.*`.

4.1.5 Composite Transfer Function

As we have defined the transfer function for each individual security extension, we combine them together to build the composite transfer function. The composite function covers two chains of transfer functions for the implicit and explicit intent space, respectively. To build each chain of transfer functions, we start from integrating the transfer functions of those security extensions that restrict an app from *sending* intents. Then, the transfer functions of the security extensions that restrict an app from *receiving* intents follow. For the transfer functions defined in this section, their composite transfer function T is defined as:

$$T(a,i) = \begin{cases} T_{PERM}^{I}(T_{IF}^{I}(T_{IFW}^{I}(T_{PB}^{I}(a,i)))) & \text{if } i \subset \mathcal{I}_I \\ T_{PERM}^{E}(T_{IFW}^{E}(a,i)) & \text{if } i \subset \mathcal{I}_E \end{cases}$$

[5]ANDROID_SDK_ROOT/platforms/android-19/data/broadcast_actions.txt

5. EVALUATION

In this section, we first discuss a prototype implementation of IntentScope. We then present the experiments in which we apply IntentScope to check intent-based communication mediated by the AOSP security extensions installed in commodity Android devices and customized Android OSs. We conclude with an evaluation of the throughput of our system.

5.1 Implementation

IntentScope includes an implementation of the intent space model, a watchdog app that monitors and incrementally acquires the policies of the AOSP security extensions, a set of policy parsers that build and compose transfer functions, and a graph builder that converts the composite transfer function into the holistic reachability graph.

The intent space model is built on Augeas Libfa [1], a native library that supports accurate and fast operations on regular expressions. In particular, we opt for Hopcroft's DFA minimization algorithm [22] to minimize regular expressions. This algorithm runs in $O(nlogn)$ time in the worst case, where n is the number of states of a regular expression's equivalent DFA. The watchdog app runs as a privileged system app. It detects state changes in PMS/AMS triggered by app installs/uninstalls and re-acquires the intent filters and permissions, regardless of whether they are statically declared in apps' manifest or dynamically registered in app's code. The watchdog app also fetches the relevant files where IntentFirewall and protected broadcasts store their policies. As the operations over intent spaces are both computation and memory intensive, the parsers and graph builder run on a dedicated server rather than on the mobile device where the watchdog app runs.

5.2 Experimental Setup

We evaluated IntentScope on two Android devices and four Android-based OSs, as shown in Table 1. The Galaxy Note ran Samsung's deeply customized Android (4.4.2), which pre-installed a large number of Samsung's apps. The Nexus 4 ran three OSs, including stock Android (5.0), MIUI (4.4.2), and CyanogenMod (4.4.4). We kept them as they were and did not install additional apps. In particular, the first two OSs pre-installed a few proprietary Google-branded apps. MIUI and CyanogenMod did not include these apps due to licensing restrictions.

For each OS, we started each installed app and kept it in the foreground. After the apps were started and IntentScope's watchdog app did not report any new policy updates in the latest one minute, we applied IntentScope to generate a reachability graph G and two subgraphs G_I and G_E that respectively represent the holistic forwarding state of

Table 1: Evaluated Android Devices/OSs and Generated Reachability Graphs

	Device	OS	$\|V\|$	$\|E_I\|$ $\|E_E\|$	Global Clustering Coefficient	Standard Deviation
1	Samsung Galaxy Note II	Customized Android	311	880,456 979,993	0.986 0.994	0.007 0.006
2	LGE Nexus 4	Stock Android	108	155,369 138,651	0.971 0.990	0.014 0.009
3		MIUI v5	104	99,170 118,707	0.979 0.991	0.013 0.009
4		CyanogenMod 11 M12	85	38,606 47,458	0.974 0.989	0.015 0.011

Table 2: Apps Ranked by PageRank

	Highest in G_I	Lowest in G_I	Highest in G_E	Lowest in G_E
1	**com.viber.voip** com.android.contacts com.android.settings	com.android.proxyhandler com.monotype.android.font.cooljazz com.sec.android.provider.badge	com.android.contacts com.android.phone com.android.settings	com.sec.enterprise.permissions com.samsung.android.mdm com.samung.android.sdk.spenv10
2	com.google.android.apps.plus com.android.settings com.google.android.apps.gms	com.android.dreams.basic com.android.providers.userdictionary com.android.vpndialogs	**com.google.android.setupwizard** com.google.android.apps.plus com.android.settings	com.android.dreams.basic com.android.wallpaper com.google.android.apps.docs.editors.slides
3	com.android.mms com.android.contacts com.android.settings	com.android.pacprocessor com.android.sharedstoragebackup com.miui.providers.weather	com.android.email com.android.mms com.android.settings	cm.android.printspooler com.android.nfc com.android.noisefield
4	com.android.gallery3d com.android.email com.android.contacts	com.android.nfc com.android.backupconfirm com.android.sharedstoragebackup	com.android.contacts com.android.email com.android.settings	com.android.nfc com.android.incallui com.android.printspooler

implicit and explicit intents. Each vertex represents an app identified by its package name rather than UID[6]. Parallel edges are allowed and prevalent in the graphs to capture the multiple entry points of an app.

Table 1 lists the number of vertices, the number of edges (including parallel edges), and the global clustering coefficient (measured without parallel edges) of each G_I and G_E. A global clustering coefficient is a measure of the degree to which vertices in a graph tend to cluster together. We opted for this measure to get a general idea about how freely the installed apps on a mobile OS are allowed to communicate with one another. As the clustering coefficient of a clique is 1, the measured values of C_G indicate that the vertices in all the graphs are densely connected, which is in line with our observation that most apps have at least one component (the main activity) exposed to other apps. The large number of edges also implies the complexities of managing fine-grained policies for intent-based communication.

Given the large number of apps/vertices and edges, prioritizing the apps that expose larger attack surfaces is critical for efficiently analyzing and resolving policy conflicts and violations. Therefore, we propose to identify such apps with PageRank [30]. The underlying intuition is that such apps are more likely to be accessed by other apps and thus have more incoming edges, and the apps that have direct incoming edges from such apps are also likely to be attacked. Table 2 lists the apps in the four mobile OSs with the highest and lowest rankings. Most of the listed apps are in line with intuition, such as `com.android.settings` and `com.android.email`. Here we discuss two apps which are displayed in bold in Table 2. The app `com.google.android.setupwizard` is highly ranked because it exports 69 components that can be accessed with explicit intents. The app `com.viber.voip` is highly ranked because of its 94 intent filters that expose the components to implicit intents.

[6]Apps with the same UID are considered as separate apps but share the permissions of one another [9].

5.3 Experiments

With IntentScope, checking what intents an app can send is equivalent to checking the vertex's outgoing edges as well as the intent spaces assigned on them. Conversely, checking what intents an app can receive is equivalent to checking the incoming edges. In addition, IntentScope supports flexible queries backed by regular expressions. Next we elaborate four experiments in which we leverage the insights provided by IntentScope to identify potential vulnerabilities due to errors in security policies of the AOSP security extensions.

5.3.1 *Zero Permission ≠ Zero Privilege*

Enforcing least privilege is a common practice in mobile security. While recent work [14, 27, 35] attempts to control and minimize the set of an app's granted permissions, we are interested in another question: *what can an app do if it has no permissions*. In this experiment, we created and installed such a *zero-permission app*. We then checked what components this app can reach with its allowed intents. This experiment helps a policy analyst reveal the exposed components that could possibly be exploited by even a zero-permission app. If any sensitive components are exposed, the details of the allowed intents that reach these components provide the necessary knowledge for a policy analyst to create precise policies that protect them. We find that zero permission does not necessarily mean zero privilege as users might expect. Table 3 shows the number of the zero-permission app's reachable apps (*i.e.* out-neighbors) and its local clustering coefficient.

The flexible queries supported by IntentScope also allow a policy analyst to pinpoint the intents that have interesting semantics. In the Galaxy Note, we found that this zero-permission app can send implicit intents that contain an interesting scheme called `android_secret_code`. For example, one of the reachable apps is `com.sec.android.app.wlantest`, which accepts intents with an action `android.provider.Telephony.SECRET_CODE`, an authority of `526`, and a scheme

Table 3: Reachability of a Zero-Permission App

	# Outgoing Edges	# Reachable Apps	Local Clustering Coefficient
1	2,767	241	0.943
	3,072	263	0.968
2	1,443	77	0.905
	1,280	92	0.960
3	955	79	0.927
	1,142	90	0.968
4	454	62	0.914
	557	72	0.961

of `android_secret_code`. Another reachable app `com.wssyncmldm` is a sensitive app that can silently download and install apps. Therefore, an app with no permissions could exploit a vulnerability in this app in order to download and install apps, thus escalating the privilege of the zero-permission app without exploiting the underlying OS. We also found that a recent attack [31] is applicable here, where a malformed intent sent from a zero-permission app can exploit and take over the exposed sensitive app.

5.3.2 Fine-grained Domain Isolation

Chin *et al.* [13] presents a limitation of intent-based communication. Suppose a malicious app Mallory attempts to attack a legitimate and sensitive app Alice and existing policies prevent their direct communication. The limitation allows Mallory to eavesdrop the intents from Alice to Bob and allows Mallory to send spoofed intents to Alice. This situation calls for a fine-grained domain isolation model that not only considers apps but also includes intents. IntentScope is useful as it provides insights about intents.

Specifically, two apps are not isolated with respect to eavesdropping attacks if they share in-neighbors and incoming intents in the reachability graph. They are not isolated with respect to spoofing attacks if they share out-neighbors and outgoing intents. Thus, IntentScope guarantees intent isolation between two apps if: (1) the apps are not neighbors of each other; *and* (2) the intent spaces of their incoming edges from common in-neighbors do not intersect; *and* (3) the intent spaces of their outgoing edges to common out-neighbors do not intersect.

As a case study, we checked the intent isolation between two apps in the Galaxy Note: `com.android.externalstorage` and `com.fmm.dm`. The former is an Android system app. The latter is believed to be bloatware as reported on several online forums. IntentScope reported that the intent spaces do not intersect, which implies that no app steals any intent from the other. However, these two apps share 242 common out-neighbors and the intersection of the intent spaces is not empty (see Figure 3(a)). Therefore, these apps are still susceptible to spoofing attacks.

5.3.3 Enumerating Multi-app Workflows

In modern mobile operating systems, it is common for a user to orchestrate multiple apps for a large and user-defined task. For example, a user may streamline a workflow of downloading, viewing, editing, and sending a picture with a chain of apps. Under the hood of Android, a multi-app workflow is implemented as a calling sequence of intents. While controlling such workflows has been well covered by Aquifer [26], IntentScope provides clues for a policy administrator to create precise rules that can be enforced by Aquifer and similar access control systems.

In this experiment, we applied IntentScope to enumerate the workflows in MIUI that match the aforementioned example. Specifically, we started from an app `com.android.providers.downloads`, which manages downloaded files. We then performed a breath-first search on the reachability graph for a sequence of implicit intents as follows:

1. action=`android.intent.action.VIEW`, scheme=`content`, category=`android.intent.category.BROWSABLE`;
2. action=`android.intent.action.EDIT`, type=`image/*`;
3. action=`android.intent.action.SEND`, type=`image/*`.

Figure 3(b) shows the matching workflows that start from the cyan node. The grey nodes are the first hop; the purple nodes in the middle are the second hop. Note that the purple nodes also serve as the first hop because the photo editors can also handle the `VIEW` action. The yellow nodes represent the last hop where data may leave a mobile device via emails, Bluetooth, or MMS messages.

5.3.4 Discovering Permission Re-Delegation Paths

A zero-permission app may send an intent to a privileged app, thus delegating the privileged app to perform permission-protected tasks for it [19]. In other words, permission re-delegation happens when apps with respective permission sets communicate with each other with intents. Under this definition, existing work [10, 19] detects and mitigates permission re-delegation attempts at runtime. One step further, we expect to enable a policy analyst to get insights into potential permission re-delegation paths before apps may execute. Meanwhile, the intents used along re-delegation paths provide semantics for the policy analyst to make informed decisions and take precise actions against the privileged apps that could be abused.

We propose to use *connected subgraphs* to represent permission re-delegation paths in a reachability graph. A subgraph is connected if every pair of its vertices has a path that consists of *only* the vertices in the subgraph. This is analogous to the situation where multiple apps collude but cannot relay their communication via other apps. We define the problem of discovering re-delegation paths as follows: given a set of critical permissions denoted as CP, find all the connected subgraphs of k vertices that satisfy:

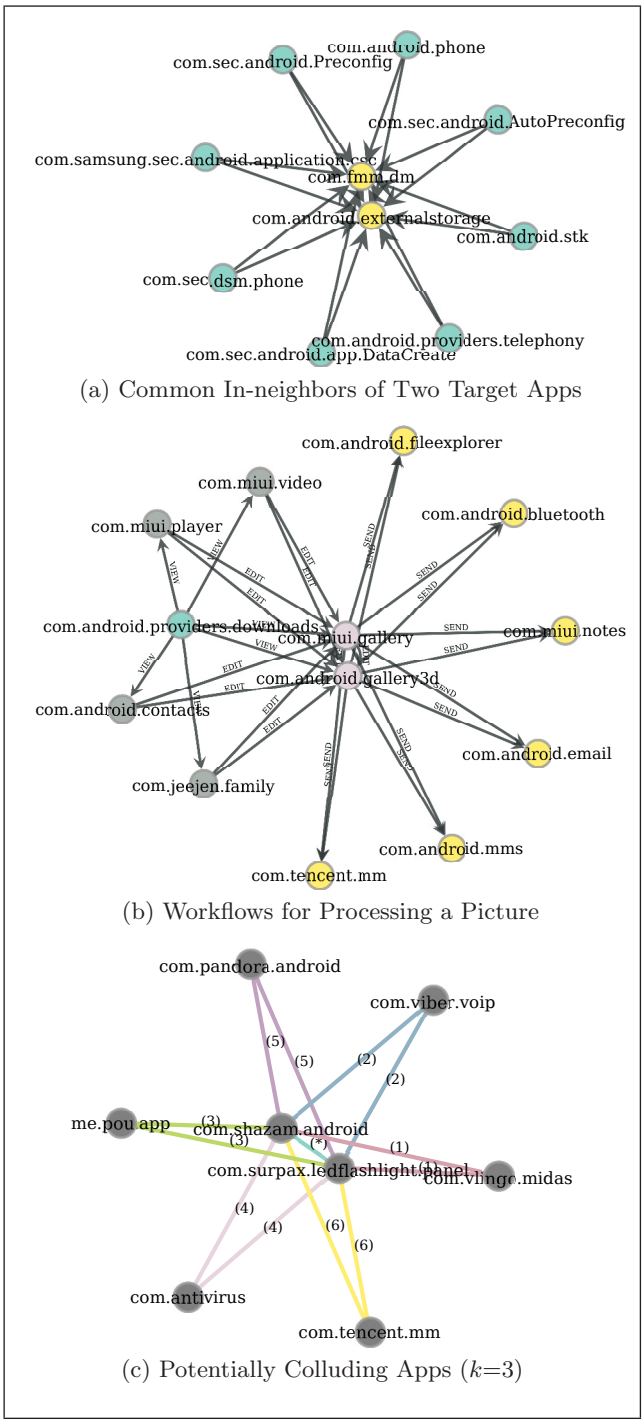
- Each app (vertex) holds at least one permission but not all the permissions in CP; and
- The union of the apps' permissions is a superset of CP.

The best algorithm we found to generate connected subgraphs of k vertices is $ConSubG(G, k)$ [24], whose worst-case time complexity is exponential in k. The performance of this algorithm is generally acceptable because we rarely encounter cases where more than five apps collude.

We targeted the third-party apps installed on the Galaxy Note and set $k = 3$. We attempted to create a synthetic attack where apps collude to drain the battery with a critical permission set of three permissions: `BLUETOOTH_ADMIN`, `NFC`, and `FLASHLIGHT`. Our results show 6 groups of apps (triangles) that can possibly collude to cover the critical permissions. In particular, the two apps in the center respectively hold `FLASHLIGHT` and `NFC`, while the surrounding six apps hold `BLUETOOTH_ADMIN` (see Figure 3(c)). After the groups

Table 4: System Throughput

	$\|E_I\|$	Avg. Time (s)	StdDev (s)	# edges/sec	$\|E_E\|$	Avg. Time (s)	StdDev (s)	# edges/sec
1	800,456	302.05	5.73	2,915	979,993	115.57	2.02	8,454
2	155,369	70.08	3.02	2,217	138,651	21.59	0.74	6,422
3	99,170	38.69	0.92	2,563	118,707	16.92	1.02	7,014
4	38,606	15.63	1.00	2,469	47,458	6.77	0.45	7,013
Average				2,541				7,225



(a) Common In-neighbors of Two Target Apps

(b) Workflows for Processing a Picture

(c) Potentially Colluding Apps (k=3)

Figure 3: Experimental Results

are identified, a security analyst can further look into the apps for colluding behaviors with static or dynamic analysis. On the contrary, a user can eliminate colluding attacks by placing the apps into separate domains.

Even though the discovered eight apps are mostly downloaded and seem to be trusted by general users, they may carry third-party libraries or vulnerable components that are exploitable by other apps. In other words, they may not deliberately collude, but could be exploited by other apps to acquire privileges. The analysis discussed in this experiment can be combined with the other analyses (*e.g.* zero-permission apps) to further generate knowledge for a policy analyst to take precautions before real exploits occur.

5.4 System Throughput

To understand the performance of IntentScope, we performed a microbenchmark to evaluate the number of edges that IntentScope can check in a second. Given that checking an edge is done by testing whether the intersection of the edge's intent space and a given intent space is empty, this benchmark also implies the throughput of IntentScope in terms of processing intent spaces. In the benchmark, we used the following two intent spaces to evaluate the throughput of implicit intents and explicit intents, respectively. Note that the intersection of an implicit intent space and an explicit intent space is always empty and thus not evaluated.

- i_I: action=`android\.intent\.action\.EDIT`, category=`android\.intent\.category\.DEFAULT`, scheme=`http`, authority=`\d+`, type=`mpeg`, permission=`.*`;
- i_E: component=`com\.sec\..*`, permission=`.*`.

We performed the benchmark in a Xen VM running Ubuntu 14.04 with Intel Xeon E5620 2.4GHz and 8GB of RAM. Only one core was used during the benchmark. Table 4 shows the average results of 10 runs. It took approximately 5 minutes to check the customized Android OS of the Galaxy Note loaded with 311 apps, and less than 1 minute to check the others. In general, the processing time is proportional to the number of edges. As shown in Table 4, IntentScope processed 2,541 implicit intent spaces and 7,225 explicit intent spaces in a second. While explicit intent spaces were almost three times faster than implicit intent spaces, we note that an explicit intent spaces has only two dimensions and an implicit intent space has six dimensions.

6. DISCUSSION

Policy analysis and app analysis. In terms of providing insights for configuring security extensions, our intent space based policy analysis complements existing static and dynamic app analysis. We make this argument based on the fact that an app's runtime behaviors on a specific mobile device are shaped by (1) the app whose code specifies

its executional semantics; and (2) the security extensions whose policies specify how the app's specific behaviors are restricted. While we admit that app analysis is indispensable, we also note the alarming trend of malware thwarting app analysis. For example, code obfuscation and encryption hide an app's true semantics from static analysis. "Split personalities" in apps [8, 23] make malware appear innocent by detecting and evading dynamic analysis tools. To get an upper hand against adversaries, we would need policy analysis to orchestrate security extensions.

Generality of intent space analysis. While we presented intent space analysis for checking intent-based communication, the underlying methodology is beyond the scope of intents and generally applicable to other security extensions. A promising target is SE Android [32], which controls almost every inter-application communication mechanism other than intent-based communication. Specifically, it checks an attribute called *security context* when an app requests to access files, sockets and so on. Given that security contexts and intent attributes are essentially *access control labels* [16], we foresee that our intent space analysis can be extended to a "context space analysis" for SE Android. We will extend our framework to reason about SE Android policies and further maximize the coverage of inter-application communication. However, we also admit the limitation that the current intent space analysis cannot directly work with existing context-aware security extensions. As for future work, we shall map contexts into dynamic policy and provide support for such extensions.

Usability of the holistic reachability graph. As we focused on developing the intent space model and implementing a prototype of IntentScope, usability of the reachability graph was not the primary goal. Indeed, policy verification is a complicated task because the number of apps and the allowed intents among them can be quite large. However, policy management is inevitable to validate policy-driven security extensions. IntentScope attempts to reduce the burden on policy analysts by helping them intuitively perform intent-based communication analysis and utilize flexible queries. Moreover, we believe that the usability of the graph has a lot of space to improve and indeed this is an important research challenge to explore. For example, the more interactive visualization may assist a security analyst in understanding the inter-application communication and in ultimately developing a robust security policy.

7. RELATED WORK

Static and dynamic app analysis. App-oriented analysis provides insights for a policy analyst to create appropriate security policies. ComDroid [13] is the first work that discusses the intent-based attack surfaces and discovers vulnerable components mistakenly exported by apps. CHEX [25] is also built on static analysis that comprehensively discovers vulnerable ICC entry points in addition to just exported components. Epicc [28] checks ICC vulnerabilities based on a sound and detailed ICC model and scales well. AmanDroid [34], FlowDroid [6], and DroidSafe [20] statically discover information flows that potentially leak sensitive data. Elish *et al.* [17] statically reconstruct intents among apps to detect collusion. Beyond static analysis, dynamic runtime solutions reveal how apps communicate through intents in real time. IPC Inspection [19] automatically reduces an intent sender's effective permissions to mitigate unauthorized privilege escalations. QUIRE [15] provides provenance of intents so that a callee can track down the original caller. XManDroid [10] maintains a system-centric call graph for the intents that have been sent and received. TaintDroid [18] and VetDroid [37] track sensitive data shared among apps with dynamic taint analysis. Along these lines, our intent space analysis assists policy analysts by systematically analyzing how security extensions confine apps' behaviors. Its analysis is based on a holistic call graph and data-flow graph derived from the intent forwarding states of security extensions in an Android device.

Experimental security extensions for Android: Besides intent filters, permissions, IntentFirewall, and protected broadcasts covered in this work, previous research has proposed a series of experimental security extensions for Android. Saint [29] and TISSA [38] support policy-driven access control for intents. CRePe [14] and APEX [27] enable context-aware and fine-grained permissions. FlaskDroid [12] and SE Android [32] are generic and flexible MAC systems that provide comprehensive protection on both Android's middleware and kernel layers. Aquifer [26] enforces distributed information flow control over intent-based UI workflows. Android Security Module (ASM) [21] and Android Security Framework (ASF) [7] provide programmable interfaces that promote the creation of customized security extensions. IntentScope facilitates defining and verifying security policies for these security extensions. It is especially useful for ASM and ASF that may host security extensions from multiple stakeholders.

8. CONCLUSION

In this paper, we have presented intent space analysis for intent-based communication. Intent space analysis is based on an intent space model and a systematic policy checking framework called IntentScope. The intent space model maps a security extension's functionality of forwarding intents as transformation on a geometric space. Based on the intent space model, IntentScope acquires the live states of multiple security extensions and further derives a holistic view that supports formal verification. Also we have described a prototype implementation, along with extensive evaluation results of our approach.

Acknowledgements

This work was partially supported by the grants from Global Research Laboratory Project through National Research Foundation (NRF-2014K1A1A2043029) and the Center for Cybersecurity and Digital Forensics at Arizona State University. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the funding agencies.

9. REFERENCES

[1] Finite automata. http://augeas.net/libfa/, 2014. Accessed: 06/2015.

[2] Bound services - Android developers. http://developer.android.com/guide/components/bound-services.html, 2015. Accessed: 06/2015.

[3] Requesting a shared file - Android developers. http://developer.android.com/training/secure-file-sharing/request-file.html, 2015. Accessed: 06/2015.

[4] Selinux policy analysis tools. https://github.com/TresysTechnology/setools, 2015. Accessed: 06/2015.

[5] Selinux wiki. http://selinuxproject.org/page/NB_SEforAndroid_1, 2015. Accessed: 06/2015.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Notices*, volume 49, pages 259–269, 2014.

[7] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. Android Security Framework: Extensible multi-layered access control on Android. In *Proceedings of the Annual Computer Security Applications Conference*. ACM, 2014.

[8] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient detection of split personalities in malware. In *Proceedings of Network and Distributed System Security Symposium*, 2010.

[9] D. Barrera, J. Clark, D. McCarney, and P. C. van Oorschot. Understanding and improving app installation security mechanisms through empirical analysis of Android. In *Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 81–92. ACM, 2012.

[10] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *Proceedings of the Symposium on Network and Distributed System Security*, 2012.

[11] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A. Sadeghi, and B. Shastry. Practical and lightweight domain isolation on Android. In *Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 51–62. ACM, 2011.

[12] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 2013.

[13] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 239–252. ACM, 2011.

[14] M. Conti, V. T. N. Nguyen, and B. Crispo. Crepe: Context-related policy enforcement for Android. In *Information Security*, pages 331–345. Springer, 2011.

[15] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 2011.

[16] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 17–30. ACM, 2005.

[17] K. O. Elish, D. D. Yao, and B. G. Ryder. On the need of precise inter-app icc classification for detecting Android malware collusions. In *Proceedings of IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy*, 2015.

[18] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems*, 32(2):5, 2014.

[19] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 2011.

[20] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow analysis of Android applications in droidsafe. In *Proceedings of the Symposium on Network and Distributed System Security*, 2015.

[21] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi. Asm: A programmable interface for extending Android security. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 2014.

[22] J. E. Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Education, 1979.

[23] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu. Morpheus: automatically generating heuristics to detect Android emulators. In *Proceedings of the Annual Computer Security Applications Conference*, pages 216–225. ACM, 2014.

[24] S. Karakashian. An Implementation of An Algorithm for Generating All Connected Subgraphs of a Fixed Size. Software (Version Oct2010), Constraint Systems Laboratory, University of Nebraska-Lincoln, Lincoln, NE, 2010.

[25] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 229–240. ACM, 2012.

[26] A. Nadkarni and W. Enck. Preventing accidental data disclosure in modern operating systems. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1029–1042. ACM, 2013.

[27] M. Nauman, S. Khan, and X. Zhang. Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, pages 328–332. ACM, 2010.

[28] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in Android with epicc: An essential step towards holistic security analysis. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 2013.

[29] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. *Security and Communication Networks*, 5(6):658–673, 2012.

[30] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[31] O. Peles and R. Hay. One class to rule them all: 0-day deserialization vulnerabilities in Android. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[32] S. Smalley and R. Craig. Security enhanced (se) Android: Bringing flexible mac to Android. In *Proceedings of the Symposium on Network and Distributed System Security*, 2013.

[33] R. Wang, W. Enck, D. Reeves, X. Zhang, P. Ning, D. Xu, W. Zhou, and A. M. Azab. EaseAndroid: Automatic policy analysis and refinement for security enhanced Android via large-scale semi-supervised learning.

[34] F. Wei, S. Roy, X. Ou, et al. AmAndroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014.

[35] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android permissions remystified: A field study on contextual integrity. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 2015.

[36] C. Yagemann. Intent firewall. http://www.cis.syr.edu/~wedu/android/IntentFirewall/index.html, 2014. Accessed: 06/2015.

[37] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in Android apps with permission use analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 611–622. ACM, 2013.

[38] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming information-stealing smartphone applications (on Android). *Trust and Trustworthy Computing*, pages 93–107, 2011.