# Technical Report: Towards Unified Authorization for Android*

Michael J. May[1]** and Karthik Bhargavan[2]

[1] Kinneret College on the Sea of Galilee, DN Emek Hayarden 15132, ISRAEL
[2] INRIA

**Abstract.** Android applications that manage sensitive data such as email and files downloaded from cloud storage services need to protect their data from malware installed on the phone. While prior security analyses have focused on protecting system data such as GPS locations from malware, not much attention has been given to the protection of application data. We show that many popular commercial applications incorrectly use Android authorization mechanisms leading to attacks that steal sensitive data. We argue that formal verification of application behaviors can reveal such errors and we present a formal model in ProVerif that accounts for a variety of Android authorization mechanisms and system services. We write models for seven popular applications and analyze them with ProVerif to point out attacks. As a countermeasure, we propose Authzoid, a sample standalone application that lets applications define authorization policies and enforces them on their behalf.

## 1 Introduction

The Android operating system seeks to foster a rich ecosystem of third-party applications. Users may download apps from reputable stores managed by Google and Amazon or directly from app developers.[3] This leaves users vulnerable to malware masquerading as genuine apps. Consequently, Android provides strong runtime isolation, running each application process in a separate Dalvik virtual machine, and giving each a private storage area. Isolated applications may still share files and data, for example using external storage or using an inter-app messaging mechanism called *intents*. While some apps freely share and collaborate with others, those holding sensitive data are tempered by the need for security and integrity. Android therefore provides authorization mechanisms which let an app control which other apps, if any, can read or write its data.

*System Permissions* Android protects its system resources through *permissions* which are granted by the user at installation time and accompany the app throughout its lifetime. The Android SDK defines about 130 built-in permissions of which some forty are *signature/system* permissions and are reserved for

* Expanded from publication in ESSoS 2013
** Work performed while visiting INRIA.
[3] It is estimated that the average Android phone in 2012 has 32 apps installed [24].

the operating system or apps installed by the device manufacturer [28]. The rest can be requested by an app in its *application manifest*, an XML file which is prepared by the developer and stored in its application package (APK) file. When an app attempts to access a system API function at run time, Android first checks if the requestor has the required permission. If it doesn't, a security exception is thrown.

Some examples of regular permissions are: `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE` permissions to read or write to the phone's shared disk (referred to informally as the *SD card* since its default mount point is `/mnt/sdcard/` [21]), `INTERNET` to open network sockets, and `READ_LOGS`[4] to access the system log. Some examples of system permissions are: `INSTALL_PACKAGES` to install new applications, `BRICK` to disable the phone completely, and `DELETE_CACHE_FILES` to clear the cache directories of other apps.

*Application-level Authorization* In addition to install-time permissions, Android provides a variety of other authorization mechanisms. Activities can filter which intents will be directed to them based on the intent's content or requested action. Content providers and services can be made available only to applications which have certain permissions. Authorization may seem seamless to the user, but due to the variety of tools available and the details of the OS, it can be technically messy and sometimes can even be bypassed.

Consider, for example, a user who installs Dropbox (`www.dropbox.com`), uses it to download a PDF file from the cloud, and opens it with Adobe Reader (`adobe.com/products/reader.html`).[5] The user would assume that during the transaction Adobe Reader got temporary read access to the file and nothing more. As we discuss later, that is not the case at all: Adobe Reader and (up until API level 16) *all* of the applications on the phone can read the file indefinitely afterwards. Many can modify it too.

*Our Contribution* There are many ways in which applications get authorization wrong or fail to enforce authorization properly. They fail primarily because they don't define the policy they are trying to enforce and (likely) didn't use a full model of the environment during testing. Proper modeling of the environment and the application's behavior would reveal attacks on the authorization mechanism.

Our contribution in this work is threefold. First, we present a unified picture of the Android authorization tools, something not previously presented in a single work. Second, we show how many popular sharing applications on the market fail to get authorization right and publish a formal model of the Android authorization tools and environment which allows us to reveal attacks. We publish the model so that others can use and extend it to test their apps. Third, we present Authzoid, a sample authorization app which properly implements the authorization tools that the apps got wrong. The code can act as

[4] Changed to signature/system/development permission in API level 16.
[5] The most popular PDF reader on Google Play as of Oct 2012.

a source code module to be included as is or as a starting point for developers who want to get authorization right. Both code and model are found at: `http://prosecco.gforge.inria.fr/Essos/pv/`.

The rest of this paper is organized as follows. Section 2 explains the Android authorization tools. Section 3 discusses our attacker model, gives technical descriptions of the sharing applications surveyed, and explains the attacks against them. Section 4 contains our formal model for Android authorization tools, environment, and the applications studied. Section 5 discusses the Authzoid app and its major features. Section 6 contains related work and Section 7 concludes.

## 2 Authorization for Android Applications

Android applications are composed of four kinds of run time entities:

**Activities** correspond to windows and allow for user interaction via a GUI.
**Content Providers** provide SQL-like interfaces to queryable data.
**Broadcast Receivers** listen for broadcast messages from other application components, the operating system, or other applications.
**Services** run in the background and provide long term functionality without providing a user interface.

Applications exchange messages via intents which contain a URI data field and strings, URIs, or key-value pairs in *extra* fields. They are routed by an Android component called *Binder* between the run time entities. During routing, the *Activity Manager* writes the action, sender, recipient, and data field (but not the extra fields) to the log.

Each runtime entity may use a variety of authorization mechanisms to control access to its data and functionality. In the rest of this section we review the five authorization mechanisms available and explain their usage, strengths, and weaknesses. For each mechanism, we give an example of how it is used in a popular application currently available from the Android Market. Some apps use a combination of the mechanisms below to enable a variety of user policies.

*Android Permissions* Android's SDK includes about 130 permissions, but an app may extend them with its own permissions by adding them to its manifest file. Apps can use permissions to enforce authorization in one of two ways.

First, content providers, activities, services, and broadcast receivers can specify that only applications with a certain permission may access them. For activities and services, this prevents applications without the permission from invoking or binding to them. For content providers, separate read and write permissions may be given. For broadcast receivers, it prevents the delivery of broadcasts from apps without the permission. The filtering is done automatically by Binder based on the app's manifest file (`android:permission` for activities, services, and broadcast receivers and `android:readPermission` and `android:writePermission` for content providers) or as defined programmatically if they are configured in code.

*Example:* K-9 Email (`code.google.com/p/k9mail/`) uses the custom permissions `com.fsck.k9.permission.READ_ATTACHMENT` to protect its attachments content provider. Its email messages content provider protects read access with `READ_MESSAGES` and write access with `DELETE_MESSAGES`. GMail (`gmail.com`) and Yahoo! Mail (`mail.yahoo.com`) (discussed below) use a similar strategy.

Second, apps can use the system API to discover whether it, the app which called it, or another app has a particular permission (using `checkPermission()` or `checkCallingPermission()`), regardless of its type. They can then make programmatic decisions based on the results.

*Example:* Some plug-in libraries (*ex.* ACRA (`code.google.com/p/acra/`)) programmatically investigate which permissions are available to their host applications before attempting actions which require particular permissions (*ex.* reading the log and sending internet data).

*URI Permissions* The content provider read and write manifest permissions give blanket read and write access. Alternatively, a content provider can give specific read or write query access to a single `content` URI under its authority. The URI permission can be granted programmatically using `grantUriPermission()` or by sending an intent to the recipient with the `FLAG_GRANT_READ_URI_PERMISSION` or `FLAG_GRANT_WRITE_URI_PERMISSION` flags set. URI permissions can be delegated by recipients. Intent-granted URI permissions are valid until the recipient app closes or is killed. Programmatically-granted ones are valid until revoked using `revokeUriPermission()`.

Binder enforces URI permissions by tracking the grants, revokes, and intents sent, so the URI does not need to be secret. Also, since intents can be routed by capability, the sender may not know which app received the permission.

Depending on whether the `content` URI refers to a database row, a file, or both, the recipient can use a *content resolver* request to read or write the corresponding rows or file. If the URI is opened as a file using `openFile()`, the content provider returns an open file descriptor for it and Binder assigns ownership of it to the recipient.

*Example:* Users can open an attachment from K-9 Email with an external viewer. When this happens, K-9 Email sends an intent to the viewer with a `content` URI for the attachment and the URI read permission flag set. The recipient can then use a content resolver to resolve the URI to an open file descriptor. GMail and Yahoo! Mail employ a similar strategy.

The use of open file descriptors leads to some technical inconveniences. First, since a file descriptor is a hard link to the file and is owned by the recipient, the sender can't close the file descriptor or delete the file until the recipient closes it. Second, if two application hold open file descriptors for the same file (*i.e.* they both requested the same URI), they cause read/read and read/write conflicts and race conditions. Third, only a few classes in the Java file API support file descriptors, making it impossible to perform random access reads

or writes to the file and making rewinding difficult. Because of these issues, some applications immediately make local copies of files passed to them by URI (*ex.* Adobe Reader) or don't enable updates to such files (*ex.* Jota Text Editor (`sites.google.com/site/aquamarinepandora/home/jota-text-editor`)).

*Private Storage* Every Android application is given its own user name, group name, and home directory. The home directories are protected by Linux file and directory permissions and by default no app can read or write the home directory of another. Apps can override the default settings to make files or directories world readable, writable, or executable using `setReadable()`, `setWritable()`, and `setExecutable()`. Then any other app can read, write, or execute the files or directories. If an app makes a file world readable in order to share it, it may include a long random string in the path to make it hard for unintended apps to guess the path name. This technique turns the path name into a secret, so the app must ensure that only the intended recipient gets the path name.

*Example:* Unlike most apps which keep all files in private storage private, Google Drive (`drive.google.com`) (discussed below) selectively sets path read and execute permissions to enable others to read files in its private storage.

*External Storage (SD Card)* Most Android devices have a shared storage space for files or data (the SD card). Read and write access to the SD card require the permissions mentioned above. Many applications (*ex.* the camera, the default browser's downloads folder) use the SD card for storing files that are either too large to keep in private storage or that are meant to be available for other apps to use. Aside from the read and write permissions, Android does not enforce access control on the SD card, so any application can read, write, or delete any file on it. Authorization can be enforced on the SD card using encryption or message authentication codes (MAC).

*Example:* The password storage app 1Password (`agilebits.com/onepassword`) stores its encrypted password database on external storage. It doesn't share passwords directly with other apps, instead using the clipboard to copy and paste passwords. Encryption protects the contents of the password database and MACs protect its integrity.

*Web Sharing* Some apps place data to be shared on a public web site and send the URL for the data to another app via an intent. Often the URL contains a long random string to make it difficult to guess, turning the URL into a secret. Another option is to protect the URL using web-based application or user authentication such as OAuth [14].

*Example:* MyTracks (`www.google.com/mobile/mytracks/`) uses GPS information to track where the device has gone, including distance traveled, speed, and elevation change. When sharing a "track" from MyTracks with another app, it first uploads a custom map to Google Maps and then sends a web URL with a long random part to the recipient via an intent.

**Summary** The authorization tools listed show the variety of mechanisms available. It's not clear if any or all of the tools are sufficient to achieve a satisfactory authorization policy. The applications that we study in the next section use different combinations of the tools to enforce authorization, but each suffer from attacks and weaknesses that demonstrate that using them correctly is not simple. In some, the tools are used incorrectly; in others, features of the Android environment defeat the app's authorization policy.

## 3 Applications and Attacks

To investigate how popular applications use the authorization tools of Section 2 to enforce their security goals, we study seven apps: two Email clients and five Cloud File Storage applications, including three specifically marketed for secure and encrypted cloud storage. We explain each application's authorization mechanism and explain how an attacker may defeat it.

### 3.1 Authorization Goals and Attacker Model

Since the apps we examine don't specify authorization policies in their documentation, we define a minimal one for the purposes of our study. Our minimal policy contains just one confidentiality rule and one integrity rule:

**Confidentiality** An app may read a file only if it owns it or if the owner and the user have authorized the reading.

**Integrity** An app may modify a file only if it owns it or if the owner and the user have authorized the modification.

The policy can be enforced by many authorization mechanisms, including those listed in Section 2.

Regarding the attacker model, we first assume that the Android protection mechanisms are enforced according to their specification (*i.e.* the phone isn't "rooted", giving arbitrary power to an app). Next, we make the same assumption that Android does regarding app isolation: that apps are mutually suspicious. The attacker is assumed to be (1) installed on the phone, (2) capable of performing polynomial time programmatic tasks, and (3) in possession of a set of authorization-related permissions that seem may seem innocuous: `READ_EXTERNAL_STORAGE`, `WRITE_EXTERNAL_STORAGE` (51% of popular applications request it), `READ_LOGS` (6% of popular applications request it[6]), and `INTERNET` (77% of popular applications request it) [10]. Any series of actions which such an attacker can take to contravene the authorization policy defined above is an attack.

[6] As of Oct 2011, `READ_LOGS` was the ninth most popular dangerous Android permission requested. In API level 16, it was converted to a system/signature/development permission, so access to it on the most recent devices is significantly reduced.

### 3.2 Study of Sharing Applications

We now consider seven popular Android applications which enforce authorization using the mechanisms defined in Section 2. The applications are chosen because they illustrate the use of a variety of mechanisms and are representative of classes of apps.

**GMail** downloads attachments to its private storage area and manages them via a content provider which is protected by custom permissions `READ_GMAIL` and `WRITE_GMAIL`. The permissions are signature level permissions, so only Google applications can request them [7]. GMail allows the user to open an attachment using an outside document viewer by sending an intent containing a URI read permission. Applications which behave similar to GMail include the built in Android Email application and K-9 Mail.

*Attacks:* The use of a protected content provider ensures that only applications sent the URI permission can read the file. However, some recipient viewers immediately make a copy of any file sent to them by URI. For example, Adobe Reader copies any file it shows to the SD card in the downloads directory, making it readable by an attacker (confidentiality).

**Yahoo! Mail** has a content provider which is protected by a custom signature permission (`com.yahoo.mobile.client.android.permissions.YAHOO_INTER_APP`). Yahoo! lets the user open attachments using URI permissions, just like GMail. However, downloaded attachments are stored on the SD card, so they are readable by any application with `READ_EXTERNAL_STORAGE`. The MailDroid (`groups.google.com/group/maildroid`) application behaves similarly.

*Attacks:* Since Yahoo! stores all downloaded attachments on the SD card, an attacker can read them (confidentiality). The application does not check for downloaded file integrity, so once on the SD card they may be modified by an attacker as well (integrity).

**Google Drive** offers two mechanisms for sharing files on the phone.

First, the on-phone app lists the files and directories on the cloud and downloads one when the user requests to view or share it. Files can't be updated on the phone. A downloaded file is placed in a new, randomly named directory in a document cache directory located in Google Drive's private storage (`/data/data/com.google.android.apps.docs/cache/`). The new directory contains just one file and is world readable and executable. The file is made world readable and all the directories above the randomly named directory are world executable, letting any app open the file, but not list the directory names under `cache`. The file path is sent to the target as the `data` field of an intent.

Second, apps can access Google Drive via a web service interface which is protected by SSL and OAuth 2.0. An app receives an API identifier which it

can use to obtain file read and write tokens. An app can download files via their names or identifiers and send updates back over the web.

*Attacks:* With respect to the on-phone app, the directory path is a secret since any application which knows it can open the file. Activity Manager, however, prints the `data` fields of all intents to the log, so the path is printed as well. An attacker which has `READ_LOGS` permission can discover the path and read the file (confidentiality).

**Dropbox** lists the files and directories in the cloud and downloads them on demand. The files are stored on the SD card in a directory called `scratch`. When sharing a file, it is first stored in the `scratch` directory and then the path and filename are sent via an intent.

Downloaded files can be opened for reading and editing, but are not checked for integrity after downloading. When opening a file, the user chooses which file viewer to use; if the viewer saves a new version, it is uploaded to the cloud. Saves are monitored until the authorized viewer closes or loses focus. Dropbox ignores saves by other applications, even when an authorized viewer is working.

When sharing a file for attachment to an email, the file is uploaded (if necessary) and a web URL is provided in the intent as an extra. The URL provides read only access to the file and includes a random string to make guessing harder.

*Attacks:* Since downloaded files are stored on the SD card, they are readable by an attacker (confidentiality). Unauthorized saves are not automatically uploaded to the cloud, but since there is no integrity check, an attacker can tamper with a file and subsequent views of the file on the phone will show the tampered version. If a viewer unknowingly saves a tampered version, the attacker's modifications will reach the cloud (integrity).

**Wuala** (`www.wuala.com/`) like Dropbox and Google Drive, downloads files on demand and like Dropbox stored them on the SD card in a directory called `edit`. Its sharing behavior, however, depends on the file type. Once downloaded, text documents are left in `edit` permanently, but audio and video files are deleted within a few seconds after the playing application has finished.

When viewing a file, its paths is sent via an intent. If a file has been shared for editing, edits by *any* other application are propagated to the cloud until the authorized application closes. Otherwise, modifications to files in `edit` are ignored. An integrity check is performed before each share. If the file in `edit` fails the check, a new copy is downloaded form the cloud.

*Attacks:* Since Wuala stores files on the SD card, they are all readable by the attacker (confidentiality). Media files are a bit harder to read than text files. An attacker can tamper with files while they are open for editing by another authorized application (integrity), but the user is shown a notice in the notification bar and so may realize that an attack has happened.

**Spider Oak** (`spideroak.com`), like the other cloud storage applications, downloads files on demand. Like Dropbox and Wuala, it stores its downloaded files on the SD card. Spider Oak, however, does not allow on-phone editing of any files. Modifications to the files in the SD card cache are completely ignored, but an integrity check is not performed before sharing to ensure that the file has not been tampered with.

*Attacks:* As with Dropbox and Wuala, its files are on the SD card and so are readable by an attacker (confidentiality). Since there is no integrity check before sharing, files may be tampered with and on-phone applications may receive tampered files (integrity).

**BoxCryptor** (`boxcryptor.com`) encrypts files stored on cloud file storage services such as Dropbox and Google Drive or on the SD card. Its older "Classic" version encrypts files using 256-bit AES in cipher block chaining mode (CBC), but uses a single initialization vector (IV) for each folder. Therefore, files in the same directory which begin similarly will have identical initial cipher text blocks. Its current version uses a different IV for each file.

Neither version uses message authentication codes (MACs) for file integrity checks. CBC's chaining dependencies mean that encrypted files can be modified and produce predictable outcomes. Both versions enable file viewing and editing on the phone. Files are downloaded from the cloud on each open request and stored in a cache directory on the SD Card. File paths are sent to recipient apps using the file's path in the data field of an intent. Once opened for editing, any app can modify the file in the cache. The modified version is encrypted uploaded to the cloud version. The current version lets the user clear the cache by request and on exit.

*Attacks:* As with Wuala and Spider Oak, the use of an offline cache on the phone leaves files visible to other apps (confidentiality). The lack of integrity checks on encrypted files and of tracking editing by apps means they can be tampered with on the phone (integrity). The modified version will be shown to other apps which attempt to open a maliciously modified file. The user is notified of any upload to the cloud of a modified file.

Using Boxcryptor to encrypt and manage Google Drive files has the consequence of making the files more vulnerable to on phone malware. Since decrypted files are stored on the SD Card and can be modified by any application, the read-only and private link protections from Google Drive are circumvented.

### 3.3 Discussion

Our study of seven popular and well-regarded applications illustrates the difficulty in getting even a simple authorization policy right. Many applications place sensitive data on public external storage. Some use unguessable directory names in private storage, but these names may leak into the shared system log.

Still others may themselves correctly implement access control, but may be let down by the applications with which they share files or which they allow to manage stored content.

Simple technical tricks aren't sufficient against a dedicated adversary. Wuala (`wuala.com`) tries to place shared files on the SD card for only a few seconds. However, due to Android's application life cycle, a malicious app can monitor the SD card and breach confidentiality during that gap. Boxcryptor encrypts files in cloud storage, but leavers them unencrypted and vulnerable in a cache. Even just in time decrypted is limited by key management, that is, how to securely transfer a secret key to the recipient. Google Drive's example shows that transferring secret keys by intent is not always secure. Keys derived from passphrases are hard to keep secret, as shown by Belenko and Sklyarov [3]. Even if encryption keys are shared securely, file storage applications often misuse encryption and integrity algorithms and expose their plaintext to attackers [4].

We advocate a unified comprehensive approach to the implementation of application-level authorization. Rather than suggest point-fixes to prevent specific attacks, we show how to write formal models that precisely capture authorization policies and relevant parts of the execution environment. By automatically analyzing such models, we can both find attacks and gain confidence in the mechanisms used to enforce the policy.

## 4 Formal Model

As shown above, implementing even a minimal authorization policy requires an analysis of the authorization tools as well as the environment. Modeling can help such an analysis by including relevant parts of the Android authorization tools and the operating system. Developers can then create a model of their application, run it inside the Android model, and use automated tools to discover attacks. In this section we describe the building of such a model using ProVerif [5]. We show illustrative snippets of its parts: (a) the authorization policy, (b) the Android authorization tools, (c) parts of the Android OS, and (d) the sharing application. We then use the model to discover attacks in the models of the applications surveyed. ProVerif is well suited for our needs since (1) it enables the definition of authorization policies using Horn clauses and communication using the applied pi calculus; (2) it can model enforcement mechanisms that use secret and fresh file or path names and cryptography; and (3) it lets us analyze the models against an unbounded adversary.

A full explanation of the Proverif model can be found in Appendix A.

### 4.1 Formal System Policy and Model

Before explaining the ProVerif code which implements the authorization policy, we define the behavior of the Android file system, permissions, and content providers. Our goal is to define the behavior implemented by Android as a basis

for implementation in ProVerif. We define Android's behavior in a series of Horn clauses.

The following snippet shows how the file system is modeled. Lines 1–2 define paths (P) as being either the file system root (/) or some concatenation of paths (P/F1). Line 3 shows that all apps (A) are aware of the file system root. Lines 4–5 show that apps can be aware of directory child objects (files or child directories) (D) in a path if the path and child object name are known or if the path is *readable*. This corresponds to a known directory being listable by an app. Line 6 shows that apps are given a private space and so may *own* paths. Line 7 shows that the SD Card's file system is readable and listable by any app which the user has granted READ_EXTERNAL_STORAGE (the permission granting mechanism is omitted).

```
1 P is /.
2 P is P/F1.
3 A knows /.
4 A knows P/D :- A knows P AND A knows D.
5 A knows P/D :- A knows P AND P is readable.
6 A knows P :- A owns P.
7 A knows P :- P is on SDCard AND User says A has READ_EXTERNAL_STORAGE.
```

The following snippet shows how communication between apps is modeled. The previous snippet showed how apps can know things about paths and files. Line 8 shows that an app A can know about an object X if it is sent information about it by another app B. This corresponds to an app sending another one information about a path or directory name via an explicit intent. Line 9 shows communication via an implicit intent where app B sends an intent to the Android system G and the user selects app A to receive the intent.

```
8 A knows X :- B knows X AND B sends X to A.
9 A knows X :- B knows X AND B sends X to G AND User says A can read X.
```

The following snippet shows how file management is modeled. Lines 10-11 state that an app A can read file F1 if it's stored at a path P that A knows, A can execute (list) the path's contents, and F1 is readable by A (see lines 18–21 below). Lines 12–13 show a parallel rule for writing with a parallel requirement for F1 being writable by A (see lines 14–17 below).

```
10 A can read F1 :- F1 is stored at P AND A knows P AND P is executable by A
11       AND P/F1 is readable by A.
12 A can write F1 :- F1 is stored at P AND A knows P AND P is executable by A
13       AND P/F1 is writable by A.
```

The following snippet shows how file permissions are modeled. App A can write a path P if it's world writable (line 14), owned by it (line 15), on the SD Card and A has the SD Card write permission (line 16), or the internet and

A has the internet access permission (line 17). App A can read a path under a similar set of circumstances (lines 18–21). The situation is similar for executing (list) a path (line 22–24) with the exclusion of the web case since the ability to list varies by the web server's internal rules.

```
14 P is writable by A :- P is world writable.
15 P is writable by A :- A owns P.
16 P is writable by A :- P is on SDCard AND User says A has WRITE_EXTERNAL_STORAGE.
17 P is writable by A :- A knows P AND P is on the web AND User says A has INTERNET.
18 P is readable by A :- P is world readable.
19 P is readable by A :- A owns P.
20 P is readable by A :- P is on SDCard AND User says A has READ_EXTERNAL_STORAGE.
21 P is readable by A :- A knows P AND P is on the web AND User says A has INTERNET.
22 P is executable by A :- P is world executable.
23 P is executable by A :- A owns P.
24 P is executable by A :- P is on SDCard AND User says A has READ_EXTERNAL_STORAGE.
```

The following snippet shows the model for the log and URI permissions. Line 25 shows that an app knows what it's in the log if it has the log access permission. Line 26 shows that an app can read a file F1 if it knows the URI permission for it. Line 27 shows the parallel rule for writing. Note that the URI rules combined with lines 8 and 9 allow for apps to delegate URI permissions.

```
25 A knows X :- X is in Log AND User says A has READ_LOG.
26 A can read F1 :- A knows UriReadPermission(F1).
27 A can write F1 :- A knows UriWritePermission(F1).
```

### 4.2 Policy Language

The snippet below implements the minimal authorization policy from Section 3.1. It allows an app to read or write files only if it is the file's owner or if it receives authorization from the owner and the user. Lines 1–2 are a Horn clause saying that if an application (a1) and a user (u) own a resource (r), then a1 is authorized to read r. Lines 3–4 enable another application (a2) to receive read authorization from the owners (a1 and u). The parallel write rules are elided, but can be see in the full model in Appendix A.

```
1 clauses forall u:Principal, a1:appid, a2:appid, r:resource;
2       owners(r, u, a1) -> readAuthorized(a1, r).
3 clauses forall u:Principal, a1:appid, a2:appid, r:resource;
4       owners(r, u, a1) && userAuthorizedRead(u, a2, r) -> readAuthorized(a2, r).
```

*Android Authorization Tools* We implement the following Android authorization tools:

**Permissions** are included via a `androidPerm` type which is populated with permissions that can be granted by the user during installation.

**URI Permissions** are included via a `uri` type which refers to a file resource. Resolution is modeled using a lookup table of `uri` and `resource` pairs.

**Private Storage** is modeled by using a `path` type which refers to a location only accessible by the owner. If the path is declared world readable, writable, or executable, others can access it too. Fresh path names may be world readable, but only can be accessed if the requestor knows the path's name.

**SD Card** is modeled as a file system process which enables the storage or retrieval of objects based on `path` and `filename` objects stored in a lookup table.

**Web Sharing** is parallel to private storage, but without the need for setting path permissions.

The following snippet shows how file and log read permissions are handled (parallel file write clauses are elided). Lines 5–7 define the Android permission type, the permission to read the SD card (externalRead), and the permission to read the log (logRead). Lines 8–9 allow an application a to read a file with name f, path p, and any file and path permissions fp and pp if (1) a has the external read permission and (2) the file is on the SD card. Lines 10–11 allow an application a to read all files in its own private space (private(a)). Lines 12–14 allow an application a to read a file in another application o's private space if its path permissions (pp) are set to world executable (isWorldExecutable) and its file permissions (fp) are set to world readable (isWorldReadable). Line 15 allows an application a to read the log if it has logRead.

```
5  type androidPerm.
6  fun externalRead() : androidPerm.
7  fun logRead() : androidPerm.
8  clauses forall a:appid, l:location, p:path, f:filename, pp:filePerms, fp:filePerms;
9   hasPermission(a, externalRead()) -> canReadFile(a, sdcard(), p, pp, f, fp).
10 clauses forall a:appid, l:location, p:path, f:filename, pp:filePerms, fp:filePerms;
11  canReadFile(a, private(a), p, pp, f, fp).
12 clauses forall o:appid, a:appid, p:path, f:filename, pp:filePerms, fp:filePerms;
13  isWorldExecutable(pp) && isWorldReadable(fp) ->
14  canReadFile(a, private(o), p, pp, f, fp).
15 clauses forall a:appid; hasPermission(a, logRead()) -> canReadLog(a).
```

The following snippet shows how URI permissions are handled in the model. Lines 16–18 implement a set membership predicate (mem). Line 19 declares a predicate allowing an application (a) to read a URI and lines 20–23 give Horn clause for the two cases when a is allowed to: if the URI is managed by a's private content provider (private(a)) or if a is on the allowed readers list (readers) which is managed by Binder. The parallel write rules are elided.

```
16 pred mem(appid,appidList) [memberOptim].
17 clauses forall a:appid, l:appidList; mem(a,cons(a,l));
```

```
18        forall a:appid,b:appid,l:appidList; mem(a,l) -> mem(a,cons(b,l)).
19  pred canReadUri(appid, uri, appidList).
20  clauses forall a:appid,p:path,readers:appidList;
21        canReadUri(a,contenturi(private(a),p),readers).
22  clauses forall a:appid,b:appid,p:path,readers:appidList; mem(a,readers) ->
23        canReadUri(a,contenturi(private(b),p),readers).
```

*Android OS Elements* We include processes for the following authorization related Android processes:

**File System** process which enables applications to read, write, and list files on the SD card based on path and file name. The file system allows access to files in private storage by the owner and by others which know the path name if the permissions are set correctly.

**Content Provider** process which enables applications to resolve URIs and thereby read or write files which they refer to.

**Binder** process which handles the granting of URI permissions, both from the owner and via delegation. Binder writes entries in the log.

**Log** process which gives permission-based read and write access to the log.

**Permission Granting** process which enables the user to grant permissions to processes.

The following snippet shows three parts of file system's code in the model. Lines 24–25 listen for file read requests (readFile) and check the application is registered. Lines 26–27 retrieve the file based on its location (l), path (p), and file name (f), check if a is able to read it, and return it to a if it is able. Lines 28–30 listen for requests to list the files in a directory path. Line 31 allows it if the path is world readable. Lines 32–25 allow an application to list all files on the SD card if the requestor has externalRead permission.

```
24  let FileSystem() = (!in (filesystem,readFile(a, l, p, f));
25     get apps(=a) in
26     get files(=l, =p, pp, =f, fp, r) in
27     if canReadFile(a, l, p, pp, f, fp) then out(return(a), r))
28   | (!in (filesystem,listFile(a, l, p));
29     get apps(=a) in
30     get files(=l, =p, pp, f, fp, r) in
31     if isWorldReadable(pp) then out (return(a), f))
32   | (!in (filesystem,listSDCard(a));
33     get apps(=a) in
34     get files(=sdcard(), p, pp, f, fp, r) in
35     if hasPermission(a, externalRead()) then out (return(a), (p, f))).
```

The following snippet shows how a content provider appears in the model. Line 37 receives a read request (readContent) on a channel (contentprovider) from an application (a) for a URI (u). Line 38 uses the tables apps and content

(definitions not shown) to look up a in the list of registered application and to look up the contents (r), authorized readers list (readers), and authorized writers list (writers) for u (table1(=a1, a2) finds the values of a2 in table1 which are paired with a1). Line 39 checks that a is authorized to read u and outputs the contents to a via an explicit intent channel (explicitintent(a)). Lines 40–42 show a parallel process for writing, with r inserted in content and overwriting the old contents (oldr) if a is authorized.

```
36 let ContentProvider() =
37 (!in (contentprovider, readContent(a, u));
38       get apps(=a) in get content(=u, r, readers, writers) in
39       if canReadUri(a, u, readers) then out (explicitintent(a), r))
40 | (!in (contentprovider,writeContent(a, u, r));
41       get apps(=a) in get content(=u, oldr, readers, writers) in
42       if canWriteUri(a, u, writers) then insert content(u, r, readers, writers)).
```

*Testing Application* We implement a single process for each sharing application. The process registers the application, specifies how files are added, and specifies how files are shared with other applications ("open with").

The following snippet shows the Dropbox application. Line 43 defines the Dropbox application id as private (not known to the attacker initially). Line 44 is the header for the process. Lines 45–46 register Dropbox in the applications table (apps, definition elided) and publish the name of its private storage (private(dropbox)) and its web storage (web(dropbox)) by sending their values on the free channel pub (definitions of private, web, and pub are elided). This simulates an attacker knowing the application's root directory and web domain, but not knowing the paths below them where files are found. Lines 47–49 define a new file's contents (r), its file name f, and its path p; assigns ownership of the file to Dropbox (line 48, using the assume function which is elided); and inserts it in the files table (definition elided) on the web (web(dropbox)) where it stays until downloaded. The noPerms() terms are used to model file and path read, write, and execute permissions. Lines 50–55 model a user opening a file. Lines 50–51 receive an openWith command and download a file from Dropbox's web space (path p, path permissions pp, file name f, file permissions fp, file contents r). Line 52–53 select an application a and authorize it to read. Line 54 stores the file on the SD card in the files table with no explicit file or path permissions (noPerms). Line 55 returns the path and file name to the requestor by an explicit intent (explicitintent(a)).

```
43 free dropbox : appid [private].
44 let Dropbox(u:Principal) =
45    (insert apps(dropbox);
46    out (pub, (private(dropbox), web(dropbox))))
47  | (!new r:resource; new f:filename; new p:path;
48    if assume(owners(r, u, dropbox)) then
49    insert files(web(dropbox), p, noPerms(), f, noPerms(), r))
```

```
50  | (!in (openWith, ());
51    get files(=web(dropbox), p, pp, f, fp, r) in
52    get apps(a) in
53    if assume(userAuthorizedRead(user, a, r)) then
54    insert files(sdcard(), p, noPerms(), f, noPerms(), r); (*readable by attacker*)
55    out (explicitintent(a), (p, f))).
```

*Discovery of Attacks* By combining the testing application's model with the environment and authorization code, we can check two types of queries in ProVerif:

1. Checks that proper authorization is reachable. ProVerif should show traces by which a user can properly authorize an app to read and write a file.
2. Checks that an attacker can't read or write files without proper authorization.

The first queries check that authorization is possible under the model. We expect to see traces of the sort: "The file is readable by the attacker if the user has sent a read URI permission to the attacker" or "A file in private storage is readable by the attacker if the application makes the file world readable, the path world executable, and sends an intent to the attacker with the path to file." Those represent valid authorization paths in the model.

The second queries ensure that there are no other ways to read or write files aside from the authorization paths defined. If ProVerif finds any such paths, they are attacks. For example, ProVerif points out that line 54 above leaks the file to the attacker since it is allowed to read files on the SD card.

## 5 Authzoid Implementation

As shown above, the proper use of the authorization tools in Android requires careful design and analysis. In this section we describe our implementation of Authzoid, an app that lets file owners define authorization policies and then enforces them on their behalf. Authzoid uses the authorization tools explained in Section 2 to enable a wide variety of policies, including ones far richer than the minimal policy defined in Section 3.1. Authzoid is useful as a sample implementation of proper authorization, fixing the mistakes of the apps discussed above and can be useful as a starting point for developers who want to get authorization right. Its source code can be found at: `http://prosecco.gforge.inria.fr/Essos/pv/`.

Authzoid offers three application-facing interfaces: file submission, policy definition, and file retrieval. It manages file versions and authorization checks internally.

*Submission Interface* Applications can submit files to Authzoid for storage using an intent with a custom action. The intent can contain a file to share or a content URI to resolve. Files can be submitted as new or as updates to existing files.

If submitted as new, Authzoid retrieves the name of the submitting application via the Android API and stores it in its private storage area. A private database indexes the files by their original file name or URI and submitting application. The new file is assigned a new version number which is returned to the submitter. The submitter may optionally include a permission as a string extra. If included, any application with the given permission may later read or modify the file (see below).

If submitted as an update, the file must be accompanied by the name of the owner, the file's original path and file name, and a version number which indicates the last version of the file the submitter saw. Authzoid first checks if the submitter is authorized to update the file (see below). If not, an authorization failure message is returned. If the update is authorized, but the version number submitted is smaller than the current version number in the database, the update is rejected with an explanatory message. Otherwise, the file is copied in to private storage and the database is updated. Authzoid generates a new version number which it stores in the database and sends it back to the submitter.

*Policy Definition* By default, the only application which can read or write a file held by Authzoid is the submitter of the original version (the owner). Owners can set policies using the following tools, all effected using intents with custom actions:

1. The owner can request a URI read permission for the file in Authzoid. Authzoid then sends the owner back an intent with a generated URI and the `FLAG_GRANT_READ_URI_PERMISSION` flag set. The generated URI is specific to a single file version. The owner can then send the intent to another application (delegation).
2. The owner can add another application to a file's access control list for read or write permission. Revocation is similar.
3. The owner can add another application to a directory's access control list for read or write permission. This is equivalent to granting read or write access to every file contained in or below the specified directory. Directory grants do not affect files submitted using a `content://` URI. Revocation is similar.
4. The owner can set a permission (a string) on a directory or file when submitting the file or at a later time. The setting of a permission on a file is the equivalent of granting read and write permission to any application with the permission. Setting read or write on a directory enables reading, writing, and listing of any files or directories in or below the directory. Revocation is similar.

Authzoid uses the Android API to determine the name of any application which sends it an intent and to determine whether a sender has a particular permission.

By default, only the application which submitted a file (its owner) can read or write it. Authzoid enables owners to share the file via URI permissions, by adding another app to the file's read and write access control list, or by retrieving

a read-only randomized path name (similar to Google Drive). Groups of apps can be added by setting a permission on the file; then any application with the permission can read or write the file.

*Retrieval Interface* An app can request a file from Authzoid by sending an intent with owner's name, the file's original path, and name. For each request, Authzoid queries its access control matrix to see if the requestor is authorized to read the file. If the read is approved, Authzoid checks if a copy of the latest version of the file is already in the cache. If not, it generates a new directory under its private `filecache` directory with a 128-bit random name and puts a copy of the file in it. The file and random directory are set to be world readable and the directory is set to be world executable. Whether new or existing, the full file path of the file are returned to the requestor using an intent with the full path in an extra.

When an application resolves a URI using Authzoid's content provider, the content provider makes a new copy of the file, opens a new file descriptor on it, deletes the file using the Java file API, and then returns the file descriptor. This prevents read/read conflicts on the file. Since the file descriptor acts as a hard link, the Android OS will preserve the contents of the file until the recipient closes the file descriptor or is killed.

Folder listings can be requested by sending the owner's name and the path via an intent. Authzoid checks its access control matrix to see if the requestor is the owner or authorized to list the directory. If authorized, a listing of all files and directories in and under the given directory is sent back via an intent as a string array extra.

Authzoid is the first Android app that provides a unified authorization service enabling file sharing between Android apps. Using ProVerif, we verified that Authzoid is secure against the class of attacks captured in our formal model. This should not be interpreted as a formal theorem however, since our model of both Android is abstract and incomplete, and may hide other attacks. Still, our analysis presents a first step towards formal security analysis for Android applications. Our models are public and may be extended for more sophisticated analysis.

*Future Work and Extensions* Two extensions to Authzoid are under development.

First, since private storage in Android devices may be limited (Android 4.1 only guarantees 350MB of private space shared between all applications [21]), we are working on extending Authzoid to keep files not currently in use on the SD card. To protect the files, they are encrypted and authenticated using a message authentication code (MAC) and stored with their owner, original path, and version number. Encryption and MAC keys are kept securely in Authzoid's *Shared Preferences*. Files would be copied into the private space as necessary and deleted when not in use (monitored by a `FileObserver`). When loading a file, its encryption, MAC, and version number would be checked against the database. If the file's version is incorrect, the user would be shown a warning message.

Second, we observe Android handles on-phone and web authentication separately. For example, in the case of Google Drive, on-phone applications can read Drive files via the on-phone Drive application, but need to connect to the internet (and have `INTERNET` permission) to modify them. Authzoid can ameliorate this imbalance by acting as a layer of indirection between file storage applications and web sites. Requesting applications can deposit their OAuth tokens with Authzoid and then use it to request and store files. If the files are available locally, the local copy is served. Otherwise, Authzoid can request or post it to the web on behalf of the application.

## 6 Related Work

Research on Android's security infrastructure includes studies on how permissions are enforced [17], used [2], and misused or attacked [10, 12, 18, 22]. Some try to secure Android applications against attackers by performing static or dynamic analysis of apps (*ex.* [16, 8, 20]). Xu, *et al.* [29] developed Aurasium, a tool that uses static analysis and code injection to detect or prevent privilege escalation attacks. Like Authzoid, Aurasium does not require modifications to the operating system. Conti, *et al.* [11] developed CRePE, a system capable of enforcing rule based context aware security policies. Naumann, *et al.* [23] extended Android permission with custom user defined constraints. None of the above work includes formal analysis or verification.

Research on formalization of the Android stack and API includes Chaudhuri [9] who gave a formal model of a subset of the Android communication system; Enck, *et al.* [15] who developed TaintDroid to track the flow of sensitive information between Android apps (extended by Shreckling, *et al.* [25] with more complicated, dynamic run time policies); and Armando, *et al.* [1] who presented a more complete model of the Android middleware using types.

With respect to formalizations for secure sharing of resources, Blanchet and Chaudhuri [6] developed a formally verified protocol for secure file sharing on untrusted storage (a tool which could be used to secure Android's SD card) and Fragkaki, *et al.* [19] gave formal typing rules to explain Android's security model. Similar to our work, Fragkaki *et al.* described Sorbet, a modification to Android which enforces secrecy and integrity properties written by app developers. In contrast, Authzoid is developed to enable the easy specification of authorization policies and relies upon existing Android mechanisms without requiring changes to the operating system.

Since many Android apps are distributed free and make money from in-app ads, work has been done to determine how ad libraries operate and whether they pose privacy or authorization risks. Dietz, *et al.* [13] developed Quire which enabled advertisers to prevent app based ad fraud. Stevens, *et al.* [27] investigated the behavior of thirteen ad libraries and showed how their requirements cause app developers to request more permissions than necessary (*permission bloat*). As a remedy to permission bloat, Shekhar, *et al.* [26] implemented AdSplit, a mechanism to separate ad libraries from individual apps.

## 7 Conclusion

Many Android apps attempt to enforce authorization policies for sharing resources, but fail due to misuse of the Android authorization tools or due to actions by external entities. We can discover authorization attacks by using ProVerif to model a relevant subset of the Android authorization tools and environment and use it to examine the behavior of sharing applications. We also describe Authzoid, an application which lets app developers specify authorization policies for sharing and enforces them using built-in Android tools. Future extensions to Authzoid include work on making an encrypted cache on the SD card and enabling it to proxy OAuth based web sharing.

## References


1. A Armando, G Costa, and A Merlo. Formal modeling and reasoning about the android security framework. In *7th Intl Sym on Trustworthy Global Computing*, 2012.
2. D Barrera, H G Kayacik, P C van Oorschot, and A Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *17th ACM Conf on Computer and Comm Security (CCS '10)*.
3. A. Belenko and D. Sklyarov. "Secure Password Managers" and "Military-Grade Encryption" on Smartphones: Oh, Really? Technical report, Elcomsoft Ltd., 2012.
4. K Bhargavan and A Delignat-Lavaud. Web-based attacks on host-proof encrypted storage. In *6th USENIX Workshop on Offensive Technologies (WOOT'12)*.
5. B Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Computer Security Foundations Workshop (CSFW'01)*.
6. B Blanchet and A Chaudhuri. Automated formal analysis of a protocol for secure file sharing on untrusted storage. In *IEEE Sym on Security and Privacy (SP '08)*.
7. Tim Bray. Recent Android app update prevents third-party apps from using com.google.android.gm.permission.READ_GMAIL. Why? `productforums.google.com/d/msg/gmail/XD0C4sw9K7U/8KwuZl0Rl68J`, 29 July 2011.
8. P P F Chan, L C K Hui, and S M Yiu. Droidchecker: analyzing android applications for capability leak. In *ACM Conf on Security and Privacy in Wireless and Mobile Networks (WISEC '12)*.
9. A Chaudhuri. Language-based security on android. In *ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS '09)*.
10. PH Chia, Y Yamamoto, and N Asokan. Is this app safe? A large scale study on application permissions and risk signals. In *WWW '12*.
11. M Conti, V Nguyen, and B Crispo. Crepe: context-related policy enforcement for android. In *13th Intl Conf on Information Security (ISC '10)*.
12. L Davi, A Dmitrienko, A Sadeghi, and M Winandy. Privilege escalation attacks on android. In *13th Intl Conf on Information Security (ISC '10)*.
13. M Dietz, S Shekhar, Y Pisetsky, A Shu, and D Wallach. Quire: Lightweight provenance for smart phone operating systems. In *20th USENIX Conf on Security*, 2011.
14. E. Hammer-Levy (ed.). *The OAuth 2.0 Authorization Protocol*. IETF, 22 Sept 2011. draft-ietf-oauth-v2-22. Work in Progress. Expires 25 March 2012.

15. W Enck, P Gilbert, B Chun, L Cox, J Jung, P McDaniel, and A Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *9th USENIX Conf on Operating Systems Design and Implementation (OSDI '10)*.
16. W Enck, M Ongtang, and P McDaniel. On lightweight mobile phone application certification. In *16th ACM Conf on Computer and Comm Security (CCS '09)*.
17. A Felt, E Chin, S Hanna, D Song, and D Wagner. Android permissions demystified. In *18th ACM Conf on Computer and Comm Security (CCS '11)*.
18. A Felt, H Wang, A Moshchuk, S Hanna, and E Chin. Permission re-delegation: attacks and defenses. In *20th USENIX Conf on Security (SEC'11)*.
19. E Fragkaki, L Bauer, L Jia, and D Swasey. Modeling and enhancing android's permission system. In *ESORICS 2012*.
20. A Fuchs, A Chaudhuri, and JS Foster. SCanDroid: Automated security certification of android applications. Technical report, U of Maryland College Park, 2009.
21. Google. *Android 4.1 Compatibility Definition*. Android Compatibility Program, 7 Sep 2012. Rev 2.
22. P Hornyack, S Han, J Jung, S Schechter, and D Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *18th ACM Conf on Computer and Comm Security (CCS '11)*.
23. M Nauman, S Khan, and X Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *5th ACM Symp on Information, Computer and Communications Security (ASIACCS '10)*.
24. NielsenWire. State of the appnation a year of change and growth in u.s. smartphones. `blog.nielsen.com/nielsenwire/online_mobile/state-of-the-appnation-\%E2\%80\%93-a-year-of-change-and-growth-in-u-s-smartphones/`, 16 May 2012.
25. D Schreckling, J Posegga, J Köstler, and M Schaff. Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. In *6th IFIP WG 11.2 Intl Conf on Inf Security Theory and Practice (WISTP '12)*.
26. S Shekhar, M Dietz, and D Wallach. Adsplit: separating smartphone advertising from applications. In *21st USENIX Conf on Security (SEC '12)*.
27. R Stevens, C Gibler, J Crussell, J Erickson, and H Chen. Investigating user privacy in android ad libraries. In *MoST 2012: Mobile Security Technologies*.
28. K Varma. Security permissions in android. Accessed 9 Oct 2012, Krishnaraj Varma's Blog, 3 Oct 2010. `www.krvarma.com/2010/10/security-permissions-in-android/`.
29. R Xu, H Saïdi, and R Anderson. Aurasium: practical policy enforcement for android applications. In *21st USENIX Conf on Security (SEC '12)*.

## A Full Documentation of ProVerif Model

The following appendix contains a line-by-line documentation of the ProVerif model described in the body of this work. The model is divided into two parts: (1) models of the Android services and mechanisms and (2) models of app behavior. They are explained in the following sections.

For ease of use the Android mechanisms are stored in a separate ProVerif library file from the app models to be analyzed. The ProVerif engine can be run on the models by running `proverif -lib android apps.pv` from the command line in the directory where the model files and the ProVerif executable can

be found. The model as a single file can be downloaded at `http://prosecco.gforge.inria.fr/Essos/pv/`.

### A.1 Android Services

The services of the Android environment are modeled using a series of types, functions, and predicates. At the end of the model, some additional functionality is defined to enable queries on the model. The full source code of the Android services model is broken into snippets for ease of reading.

The following snippet defines types which are used to identify apps in the rest of the model. In the model, each app has an app name and app id which are kept secret from the attacker. The secrecy is used to let the model checker discern well behaving apps from malware apps which leak their name and id to the attacker. We also assume that the user must be involved in the installation of apps, so the model prevents the attacker from adding apps without the user's approval.

Line 1 defines the Principal type which is used to denote the user. Since many operations on the phone are done by the user or behalf of the user, the objects of type Principal are used to denote an action which the user has performed. Line 2 defines the appname type which denotes the name of an app. The appname models the package name of an app which Android uses as an identifier in the operating system and log. Line 3 defines the appid type which is used to identify the app in a list of apps. Line 4 defines a private function which associates an app's name to its id. Since it is labeled private, the attacker can't use it to define additional app ids. Line 5 defines a predicate which may be true of an appid - that the app it refers to has been installed. Line 6 defines a clause which states that all appname values which have been created with app() refer to installed apps. Since the attacker can't use app(), we prevent the attacker from defining fake appname values which appear to be installed apps.

```
1 type Principal.
2 type appname.
3 type appid.
4 fun app(appname): appid [private].
5 pred isApp(appid).
6 clauses forall a:appname; isApp(app(a)).
```

The following snippet defines types used for *resources*, objects which represent the contents of files and URIs. Line 7 defines the type resourceId which is the id of a resource. Line 8 defines type resourceVal which is the contents a resource. Line 9 defines the type resource which is a complex type built from a resourceId, a resourceVal, a Principal (user), and an appid (owner app) using the function res() shown on line 10. res() is defined as private, so the attacker can't use it to define its own resources directly. Lines 11–13 define projection functions (reduc) which allow the extraction of a resource's resourceId r (line 11), the appid which

owns it (line 12), and the Principal which created it (line 13). Line 14 defines a predicate which declares that an atom is a resource. It is used on lines 15–16. The clauses on those lines define that a combination of a Principal, a resourceId, a resourceVal, and an appname are a valid resource provided that the predicate res() holds of the combination (line 10). Since res() is private, the attacker is not able to create combinations which are valid resources on its own. Lines 17–18 allow the definition of a new resource using a given Principal and appid, creating resources for an app on behalf of the user.

```
7  type resourceId.
8  type resourceVal.
9  type resource.
10 fun res(resourceId,resourceVal,Principal,appid): resource [private].
11 reduc forall r:resourceId,rv:resourceVal,p:Principal,a:appid; id(res(r,rv,p,a)) = r.
12 reduc forall r:resourceId,rv:resourceVal,p:Principal,a:appid; owner(res(r,rv,p,a)) = a.
13 reduc forall r:resourceId,rv:resourceVal,p:Principal,a:appid; user(res(r,rv,p,a)) = p.
14 pred isResource(resource).
15 clauses forall u:Principal, r:resourceId, rv:resourceVal, a:appname;
16       isResource(res(r,rv,u,app(a))).
17 letfun mkResource(p:Principal,a:appid) =
18       new r:resourceId; new rv:resourceVal; res(r,rv,p,a).
```

The following snippet defines predicates which model user authorization. The predicates are used by the model to check what the user has authorized and what authorization levels apps can actually achieve. The last predicate is used by the model to check if an app is assumed to be malware. Three of the predicates are defined as block, meaning they don't appear on the right side of any implications. ProVerif is allowed to assume that a block predicate holds in order to answer queries. Thus, results may be phrased in the form: "Assuming A, B is possible."

Line 19 declares a block predicate which is used to indicate that Principal granted read permission to an appid on an resource (resourceID). Line 20 is a parallel block predicate for write authorization. Line 21 is a predicate which declares that an appid has received read authorization for a resource (resourceId). Line 22 is a parallel predicate for write authorization. Line 23 is a block predicate declaring that an appid is malware.

The above predicates are used by ProVerif to find all cases where authorization has been granted (*i.e.* readAuthorized holds on a resource) and investigate the path which lead to it holding. Acceptable paths might include user authorization (see lines 26–29 below), app delegation, or malware being granted access to the resource (see lines 30–33 below). If read authorization holds via some other path, we have found an attack.

```
19 pred userAuthorizedRead(Principal,appid,resourceId) [block].
20 pred userAuthorizedWrite(Principal,appid,resourceId) [block].
21 pred readAuthorized(appid,resourceId).
```

```
22 pred writeAuthorized(appid,resourceId).
23 pred Malware(appid) [block].
```

The following snippet defines clauses related to the authorization rules shown in the previous snippet. Line 24 defines a clause declaring that the owner of a resource r is authorized to read it (see line 19 in section 4.1). Line 25 defines a parallel write clause. Lines 26–27 define a clause stating that if the user has authorized an app a2 to read a resource r, the read is authorized (readAuthorized). Lines 28–29 define a parallel write clause. Lines 30–31 define a clause stating that if a malware app a1 has been granted read authorization, it may enable other apps to gain read authorization (perhaps by copying the resource or making it world readable). Lines 32–33 define a parallel write clause.

The inclusion of lines 30–33 allow the model to consider phones with installed malware. We are interested in examining the consequences of the malware's presence on the phone. For instance, we may query what resources the malware can and can't gain authorized access to.

```
24 clauses forall u:Principal,a1:appid,a2:appid,r:resource; readAuthorized(owner(r),id(r)).
25 clauses forall u:Principal,a1:appid,a2:appid,r:resource; writeAuthorized(owner(r),id(r)).
26 clauses forall u:Principal,a1:appid,a2:appid,r:resource;
27     userAuthorizedRead(user(r),a2,id(r)) -> readAuthorized(a2,id(r)).
28 clauses forall u:Principal,a1:appid,a2:appid,r:resource;
29     userAuthorizedWrite(user(r),a2,id(r)) -> writeAuthorized(a2,id(r)).
30 clauses forall u:Principal,a1:appid,a2:appid,r:resource;
31     Malware(a1) && readAuthorized(a1,id(r)) -> readAuthorized(a2,id(r)).
32 clauses forall u:Principal,a1:appid,a2:appid,r:resource;
33     Malware(a1) && writeAuthorized(a1,id(r)) -> writeAuthorized(a2,id(r)).
```

The following snippet defines Android-specific file system mechanisms. Lines 34–35 define types for file names and locations. Lines 36–38 define three types of locations: the SD Card, an app's private space, and an app's web site. Line 39 defines a type for paths in the file system. Line 40 defines a type for a URI. Line 41 defines function contenturi() which creates a uri as a combination of a path and a location, the domain of the URI.

The function contenturi() is labeled as data since it is available to the attacker for creation and parsing (destruction). This means the attacker can create its own URIs and discover the location and path of a URI.

```
34 type filename.
35 type location.
36 fun sdcard():location.
37 fun private(appid):location.
38 fun web(appid): location.
39 type path.
40 type uri.
```

```
41 fun contenturi(location,path) : uri [data].
```

The following snippet defines a model for file permissions functionality which parallels the Linux file permissions implemented by Android. Line 42 defines a type for an object defining file or path permissions. Lines 43–46 define the permissions that can be associated with a path or file: read, write, execute, and none. The none() permission indicates the lack of any permission and is included as an artifact of the ProVerif language.

Line 47 defines the filePerms type which contains a read/write/execute permission triple similar to the one used by Linux file permissions. Line 48 defines the perms() function which builds a filePerms object from three filePerm objects (read, write, execute). The reduc clauses in lines 49–53 define how granting a read, write, or execute permission works. For instance, lines 49–50 define that SetReadable() on a file is equivalent to creating a filePerms object using the perms() function with a read() permission set. Lines 51–52 define a parallel reduc for writes. Lines 53–54 define a parallel reduc for execute. Line 55 defines the reduc function noPerms() for declaring that no permissions have been set on an object.

Lines 56–58 define predicates for checking whether a filePerms allows world read, write, or execute. The clauses on lines 59–62 define how they are used. Line 60 defines that a perms() is world readable if its first permission is read(). Lines 61–62 define parallel clauses for write and execute.

Note that clauses are predicates which either succeed or get stuck. For example, if the ProVerif engine tries to check isWorldReadable on a perms() tuple which doesn't have read() in its first position, the clause gets "stuck" and is therefore considered not applicable.

```
42 type filePerm.
43 fun read():filePerm.
44 fun write():filePerm.
45 fun exec():filePerm.
46 fun none():filePerm.
47 type filePerms.
48 fun perms(filePerm,filePerm,filePerm): filePerms.
49 reduc forall f1:filePerm, f2:filePerm, f3:filePerm;
50     setReadable(perms(f1,f2,f3)) = perms(read(),f2,f3).
51 reduc forall f1:filePerm, f2:filePerm, f3:filePerm;
52     setWritable(perms(f1,f2,f3)) = perms(f1,write(),f3).
53 reduc forall f1:filePerm, f2:filePerm, f3:filePerm;
54     setExecutable(perms(f1,f2,f3)) = perms(f1,f2,exec()).
55 reduc noPerms() = perms(none(),none(),none()).
56 pred isWorldReadable(filePerms).
57 pred isWorldWritable(filePerms).
58 pred isWorldExecutable(filePerms).
59 clauses
60     forall f1:filePerm, f2:filePerm; isWorldReadable(perms(read(),f1,f2));
```

```
61      forall f1:filePerm, f2:filePerm; isWorldWritable(perms(f1,write(),f2));
62      forall f1:filePerm, f2:filePerm; isWorldExecutable(perms(f1,f2,exec())).
```

The following snippet defines predicates used below for determining whether an app can read or write a file (lines 63–64) and read the log (line 65). Reading a file (line 63) requires consideration of which appid wants to read, in which location the file is found, what the file's path is, what filePerms hold on the path, what the filename is, and what filePerms hold on the file. Writing a file (line 64) is similar. The intuition behind the rules is that reading or writing a file requires an app to specify the file name and path to be accessed before the file system decides whether to allow it. As mentioned above in section 3, some apps (such as Google Drive) keep path names secret to prevent apps from requesting access to even globally readable or writable files.

Reading the log (line 65) requires just the of the appid in determining whether log reading is allowed since there is a unified log for all apps.

```
63 pred canReadFile(appid,location,path,filePerms,filename,filePerms).
64 pred canWriteFile(appid,location,path,filePerms,filename,filePerms).
65 pred canReadLog(appid).
```

The following snippet defines types and functions used for managing the list of apps on the phone and managing URI read and write abilities. Line 66 defines an appidList type. Line 67 defines that the nil() list is an appid list. Line 68 defines the cons() function enabling prepending to an appid list. Line 69 defines the mem() function which checks for membership in an appidList. It uses the memberOptim option which tells ProVerif the function is to be used for set membership and can be optimized accordingly. Lines 70–72 define how mem() and cons() are used. Line 71 defines that an appid a is a member of an appidList if it's at the head of the list. Line 72 defines that an appid can be removed from an appidList to check the membership of appids not at the head (b).

Line 73 defines a predicate which declares that members of an appidList can read a URI owned by an another app. Line 74 defines a parallel rule for writing URIs.

```
66 type appidList.
67 fun nil():appidList [data].
68 fun cons(appid,appidList):appidList [data].
69 pred mem(appid,appidList) [memberOptim].
70 clauses
71      forall a:appid,l:appidList; mem(a,cons(a,l));
72      forall a:appid,b:appid,l:appidList; mem(a,l) -> mem(a,cons(b,l)).
73 pred canReadUri(appid, uri, appidList).
74 pred canWriteUri(appid, uri, appidList).
```

The following snippet defines functions related to Android permissions. Line 75 defines the androidPerm type for Android permissions. Since our model is concerned with only a subset of permissions, we only define ones for reading the SD Card (line 76), writing the SD Card (line 77), accessing the internet (line 78), reading the log (line 79), and writing the log (line 80). Line 81 defines a predicate used to examine an app's permissions. It's a block predicate, so it can be assumed by ProVerif in queries.

```
75 type androidPerm.
76 fun externalRead(): androidPerm.
77 fun externalWrite(): androidPerm.
78 fun netAccess(): androidPerm.
79 fun logWrite(): androidPerm.
80 fun logRead(): androidPerm.
81 pred hasPermission(appid, androidPerm) [block].
```

The following snippet defines clauses which determine file and URI access. Line 82 defines that an app with the logRead() permission is allowed to read the log (canReadLog). Lines 83–84 define that an app can read a content URI which refers to a path p in its own private storage (*i.e.* the content URI is in a's domain). Lines 85–86 define that an app which is listed in a content URI's readers list can read the URI even if the URI refers to another apps's private space. Lines 87–88 and 89–90 define parallel clauses for writing. Lines 91–92 define that an app a with the externalRead() permission can read a file f which is in the sdcard() location (*i.e.* stored on the SD Card). Lines 93–94 define that an app a can read the files in its private space private(a). Lines 95–97 define that an app a can read a file f in another app o's private space private(o) if its path pp is world executable (traversable) and the file is world readable. Lines 98–104 define parallel clauses for writing (SD Card, own private space, world writable).

```
82 clauses forall a:appid; hasPermission(a,logRead()) -> canReadLog(a).
83 clauses forall a:appid,p:path,readers:appidList;
84       canReadUri(a,contenturi(private(a),p),readers).
85 clauses forall a:appid,b:appid,p:path,readers:appidList;
86       mem(a,readers) -> canReadUri(a,contenturi(private(b),p),readers).
87 clauses forall a:appid,p:path,writers:appidList;
88       canWriteUri(a,contenturi(private(a),p),writers).
89 clauses forall a:appid,b:appid,p:path,writers:appidList;
90       mem(a,writers) -> canWriteUri(a,contenturi(private(b),p),writers).
91 clauses forall a:appid,l:location,p:path,f:filename,pp:filePerms,fp:filePerms;
92       hasPermission(a,externalRead()) -> canReadFile(a,sdcard(),p,pp,f,fp).
93 clauses forall a:appid,l:location,p:path,f:filename,pp:filePerms,fp:filePerms;
94       canReadFile(a,private(a),p,pp,f,fp).
95 clauses forall o:appid,a:appid,p:path,f:filename,pp:filePerms,fp:filePerms;
96       isWorldExecutable(pp) && isWorldReadable(fp) ->
97       canReadFile(a,private(o),p,pp,f,fp).
```

```
98  clauses forall a:appid,l:location,p:path,f:filename,pp:filePerms,fp:filePerms;
99      hasPermission(a,externalWrite()) -> canWriteFile(a,sdcard(),p,pp,f,fp).
100 clauses forall a:appid,l:location,p:path,f:filename,pp:filePerms,fp:filePerms;
101     canWriteFile(a,private(a),p,pp,f,fp).
102 clauses forall o:appid,a:appid,p:path,f:filename,pp:filePerms,fp:filePerms;
103     isWorldExecutable(pp) && isWorldWritable(fp) ->
104     canWriteFile(a,private(o),p,pp,f,fp).
```

The following snippet defines file system, URI, and log commands an app may attempt. Line 105 defines the command type. All of the commands in the snippet include the requesting appid which is used by the file system, content provider, or log manager to decide whether to perform the command.

Line 106 defines a read file command which includes the filename to read, the path of the file, and its location. Line 107 defines a parallel write command. Line 108 defines a directory file listing command (*i.e.* ls, dir) which includes the path to list and its location. Line 109 defines a command for listing the whole contents of the SD Card. Line 110 defines a command to create a new file and includes a filename, the path to store it, the location, and a resource which represents its contents (see lines 7–18 above). The new file will be owned by the requesting app. Line 111 defines a command for reading the log. Line 112 defines a parallel log writing command. Line 113 defines a command for reading the content of a URI. Line 114 defines a parallel write command.

```
105 type command.
106 fun readFile(appid,location,path,filename): command [data].
107 fun writeFile(appid,location,path,filename,resource): command [data].
108 fun listFile(appid,location,path): command [data].
109 fun listSDCard(appid): command [data].
110 fun newFile(appid,location,path,filename,resource): command [data].
111 fun readLog(appid): command [data].
112 fun writeLog(appid,bitstring): command [data].
113 fun readContent(appid,uri):command [data].
114 fun writeContent(appid,uri,resource):command [data].
```

The following snippet defines channels used for communication between apps and the Android operating system. Channels in ProVerif are used to transfer messages between processes. The channels defined model the communication mechanisms available in the Android application program interface (API) which allow apps to communicate with each other and with the operating system's services. The channels modeled below implement some of the features of Android communication such as explicit intents and the ability to use a content resolver to communicate with a content provider.

Line 115 defines a channel open to the attacker (pub). Information which is assumed to be known by outsiders will be given to the attacker by sending it on pub. Line 116 defines a channel for sending explicit intents. The channel is

parameterized by two appids, so logically there is a (private) channel for any app to reach any other app. Since the channel is defined as private, the attacker is not able to see messages on it. Line 117 defines a private inbox channel which can be used to communicate with an app. Like explicitintent, it's parameterized with an appid, so each app can be communicated with privately. Line 118 defines the private filesystem channel for sending commands to the file system. Line 119 defines a private channel for sending read and write commands to the log. Line 120 defines a private channel for sending read or write commands to a content provider, an operation normally performed via a content resolver.

```
115 free pub:channel.
116 fun explicitintent(appid,appid) : channel [private].
117 fun inbox(appid): channel [private].
118 free filesystem: channel [private].
119 free log: channel [private].
120 free contentprovider: channel [private].
```

The following snippet defines tables used by various processes which model Android system functions. Line 121 defines a table apps of installed apps. Line 122 defines a table files with information on every file and its properties (see lines 34–41 and 42–55 above). Line 123 defines a table content of content URIs and list of apps which can read or write them (see lines 66–74 above). Line 124 defines a table syslog with the log information written by apps.

```
121 table apps(appid).
122 table files(location,path,filePerms,filename,filePerms,resource).
123 table content(uri, resource, appidList, appidList).
124 table syslog(appid,bitstring).
```

The following snippet defines a function which models some of the functionality in the Android file system. Functions in ProVerif are similar to interactive processes in that they can react to input from other entities and apps. Functions can be divided into subprocesses to enable multiple tasks to be performed in a single process. Subprocesses can be run with the ! operator which indicates that it runs with unbounded replication. The FileSystem() function defined on line 125 is divided (|) into five listening processes which run in parallel under unbounded replication. The function is run along with the other functions in the following snippets.

Lines 126–130 define a process which listens on the filesystem channel for readFile commands (see line 106 above). The command indicates which appid requested (a), the filename to read (f), the file's path (p), and the file's location (l). Line 127 checks that a is a registered app by performing a lookup (get) on the table apps (see line 121 above) with a. If a doesn't appear in apps, the get operator gets stuck and no progress can be made, ending the command. Line 128 finds the path permissions pp, file permissions fp, and the resource object r

for the file by performing a lookup on the files table (see line 122 above). If the file details don't appear, the request gets stuck and ends. Line 129 checks if a can read the file given its permissions and owner (see lines 91–97 above). If yes, line 130 returns the file's resource (r) to the requesting app a's private channel inbox(a).

Lines 131–136 define a process which listens on the filesystem channel for writeFile commands (see line 107 above). The command has the same parameters as mentioned above for readFile with the addition of a resource (r) which is the new contents of the file. As with reading, the process begins by checking that a is registered in apps (line 132). Line 133 checks that r is a valid resource (see line 14 above). Line 134 retrieves the file's properties from the files table: path permissions pp, file permissions fp, and (old) resource oldr. Line 135 checks that a can write the file given the permissions and owner (see lines 98–104 above). If yes, a new row in files is inserted with the file's properties and new resource object r (line 136).

Lines 137–141 define a process which creates a new file. The process listens (line 137) on the filesystem channel for newFile commands (see line 110 above). The command includes parameters for defining the new app which will own the new file (a), location (l), path (p), file name (f), and resource (r). Line 138 checks that a is registered in apps. Line 139 checks that r is a valid resource. Line 140 checks that the file can be written (see lines 98–104 above) with the given the location, file name, and path. If the file can be written, line 141 inserts a new row in the files table with no file or path permissions.

Lines 142–146 define a process which lists the files in a directory. The process listens (line 142) on the filesystem channel for listFile commands (see line 108 above). The command includes parameters for the requestor's appid a, the location l, and the path p to list. Line 143 checks that a is registered in apps. Line 144 checks that the location and path exist in files and retrieves the path permission pp, the file name of a file f, the file permission fp, and the file's resource r. The table read returns only a single file, but since ProVerif's engine explores all possible outcomes, multiple files in a single directory are all considered accessible. Line 145 checks if the path is world readable (see line 60 above) to see if the requestor can read the directory (list its files). If yes, line 146 returns the file name to the requesting app a's private channel inbox(a).

Lines 147–151 define a process which lists all files in the SD Card. The process listens (line 147) on the filesystem channel for listSDCard commands (see line 109 above). The command includes a parameter for the requestor's appid a. Line 148 checks that a is registered in apps. Line 149 non-deterministically chooses a file from the files table which are in the SD Card location (sdcard()), including its path p, path permissions pp, file name f, file permissions fp, and resource r. While only a single file is returned, due to the non-deterministic nature of ProVerif's proofs, any matching file in the table is considered accessible. List 150 checks that a has the required permission externalRead (see lines 76 and 81 above) to read the SD Card. If yes, line 151 returns the path and file information to the requesting app a's private channel inbox(a).

```
125 let FileSystem() =
126 (!in (filesystem,readFile(a,l,p,f));
127       get apps(=a) in
128       get files(=l,=p,pp,=f,fp,r) in
129       if canReadFile(a,l,p,pp,f,fp) then
130       out(inbox(a),r))
131 | (!in (filesystem,writeFile(a,l,p,f,r));
132       get apps(=a) in
133       if isResource(r) then
134       get files(=l,=p,pp,=f,fp,oldr) in
135       if canWriteFile(a,l,p,pp,f,fp) then
136       insert files(l,p,pp,f,fp,r))
137 | (!in (filesystem,newFile(a,l,p,f,r));
138       get apps(=a) in
139       if isResource(r) then
140       if canWriteFile(a,l,p,noPerms(),f,noPerms()) then
141       insert files(l,p,noPerms(),f,noPerms(),r))
142 | (!in (filesystem,listFile(a,l,p));
143       get apps(=a) in
144       get files(=l,=p,pp,f,fp,r) in
145       if isWorldReadable(pp) then
146       out (inbox(a),f))
147 | (!in (filesystem,listSDCard(a));
148       get apps(=a) in
149       get files(=sdcard(),p,pp,f,fp,r) in
150       if hasPermission(a,externalRead()) then
151       out (inbox(a),(p,f))).
```

The following snippet defines a model of the log's functionality. The Log() function (line 152) is similar to the FileSystem() function above in that it is an active function which reacts to input from other entities and uses unbounded replication (!). The function is divided into two listening processes which run in parallel.

Lines 153–157 define a process which reads from the log. Line 153 listens on the log channel (see line 119 above) for readLog commands (see line 111 above). The command includes a parameter indicating the requesting app a. Line 154 checks that a is registered in apps. Line 155 uses canReadLog (see lines 65 and 82 above) to check if a is allowed to read the log. If yes, line 156 non-deterministically gets a row from the syslog table, including the app b which wrote the message and the message x. Line 157 returns the log message to the private inbox(a) of the requestor.

Lines 158–160 define a process which writes to the log. Line 158 listens on the log channel for writeLog commands (see line 112 above). The command includes a parameter indicating the requesting app a and the log message to write x. Line 159 checks that a is registered in apps. Line 160 adds the message to the syslog table with the writer's appid and message.

```
152 let Log() =
153 (!in (log,readLog(a));
154       get apps(=a) in
155       if canReadLog(a) then
156       get syslog(b,x) in
157       out(inbox(a),x))
158 | (!in (log,writeLog(a,x));
159       get apps(=a) in
160       insert syslog(a,x)).
```

The following snippet defines a process which models a part of Binder, Android's inter-app communication mechanism. Similar to the previous processes, Binder() is a function which is called to execute an interactive process and uses unbounded replication (!). It contains one subprocess.

Lines 162–165 define a process which transfers explicit intents between apps. Line 162 non-deterministically retrieves two apps from the apps table, a and b. Line 163 listens on the private explicitintent channel (see line 116 above) which connects a and b. The request contains just a bitstring x which is the intent's body. Line 164 writes the intent's body and write to the log using writeLog (see lines 158–160 above) as the Activity Manager in Android does. Line 165 sends the body of the intent to the private inbox of b, the intent's recipient.

```
161 let Binder() =
162 (!get apps(a) in get apps(b) in
163       in (explicitintent(a,b),x:bitstring);
164       out (log,writeLog(a,x)); (* This line enables the Google Drive Attack *)
165       out (inbox(b),x)).
```

The following snippet defines a process which models a content provider which can be queried by a content resolver on behalf of an apps. It allows apps to read and write data via content URIs. As in the previous snippets, ContentProvider() is an interactive function divided into subprocesses which are run under unbounded replication.

Lines 167–171 define a process which enables the reading of content based on a content URI. Line 167 listens on the contentprovider channel (see line 120 above) for readContent commands (see line 113 above). The command includes parameters indicating the requesting app a and the URI to resolve for reading u. Line 168 checks that a is registered in apps. Line 169 retrieves the resource (contents) r for the URI, the list of authorized reader apps (readers), and the list of authorized writer apps (writers). Line 170 checks if a is on the list of authorized readers for the URI using canReadUri (see lines 83–86 above). If it is, line 171 sends the resource of the URI to the requestor using its private channel inbox(a).

Lines 172–177 define a parallel write process. It is similar to the read process with the exception that it receives a new resource object nr on the contentprovider channel on line 172, checks it's a valid resource (line 175), checks that the sender is authorized to write the URI (line 176), and stores the new resource in the content table (line 177).

```
166 let ContentProvider() =
167 (!in (contentprovider,readContent(a,u));
168       get apps(=a) in
169       get content(=u,r,readers,writers) in
170       if canReadUri(a,u,readers) then
171       out (inbox(a),r))
172 | (!in (contentprovider,writeContent(a,u,nr));
173       get apps(=a) in
174       get content(=u,oldr,readers,writers) in
175       if isResource(nr) then
176       if canWriteUri(a,u,writers) then
177       insert content(u,nr,readers,writers)).
```

The following snippet defines a process which models the behavior of the Android app installer. It adds apps to the apps table and creates files for them. It is similar to the previously explained processes.

Lines 179–180 define a process which creates new apps and adds them to the apps table, effectively installing them. Line 179 creates a new appname called a. Line 180 defines it is an app using app (see line 4 above).

Lines 181–185 define a process which inserts new files into the private space of an app. Line 181 non-deterministically chooses an app from the apps table. Line 182 creates a new filename called f. Line 183 creates a new path for it called p. Line 184 creates a new resource object for the file using mkResource (see lines 17–18 above). Line 185 adds the new file name and resource to the files table in the app's private space private(a), at path p, with file name f and resource r, and no path or file permissions set.

```
178 let AppInstaller(u:Principal) =
179 (!new a:appname;
180       insert apps(app(a)))
181 | (!get apps(a) in
182       new f:filename;
183       new p:path;
184       let r = mkResource(u,a) in
185       insert files(private(a),p,noPerms(),f,noPerms(),r)).
```

The following snippet defines a process which models the behavior of a malicious app, one which delivers information to the attacker and gives it access to Android services. Line 186 defines a single Principal which is used to represent the user of the phone. It will be used in the attacker proxy code and so is defined here. Lines 187–203 contain the body of the function AttackerProxy(). It is an interactive function and contains six subprocesses under unbounded replication.

Lines 188–190 define a process which sends the attacker information about malware apps, thus giving the attacker access to it. Line 188 non-deterministically chooses an a from the installed apps. Line 189 checks if the selected a is malware using the Malware clause (see line 23 above). If a is malware, line 190 sends the appname on the public channel pub, thus making it known to the attacker.

Lines 191–194 define a process which creates resources and leaks them to the attacker. This mimics the situation where malware intentionally gives others access to phone files. Line 191 listens on the pub channel for appid objects. Such objects would be sent by the previous process (lines 188–190) for all malware apps. Line 192 checks that the received a is registered in apps. Line 193 creates a new resource owned by a (seemingly) with the approval of the phoneUser. This implies that the malware app can impersonate the user's intended actions. That is, Android can't tell whether the actions of the app are truly performed by the user or by the app without the user's knowledge. Line 194 leaks the resource created (r) on pub, thus making it known to the attacker.

Lines 195–196 define a process which transfers commands to the file system on behalf of the attacker. This mimics the situation where malware gives the attacker access to the services of the file system. The process is necessary because the filesystem channel is defined as private, meaning the attacker does not have direct access to it (see line 118 above). Line 195 listens for commands on the pub channel. Line 196 directs the command to the file system using the filesystem channel.

Lines 197–198 define a process which transfers entries to the log on behalf of the attacker. This mimics the situation where malware gives the attacker the access to write to the log. Like the previous process, it's necessary because the log channel is defined as private (see line 119 above). Line 197 listens on the pub channel for commands. Line 198 directs the received command to the log via the log channel.

Lines 199–200 define a process which transfers commands to the content provider on behalf of the attacker. It is similar to the previous two processes in that it mimics malware which gives the attacker access to a private channel, in this case contentprovider (see line 120 above). Line 199 listens on the pub channel for commands. Line 200 directs the command to the content provider via the contentprovider channel.

Lines 201–203 define a process which gives the attacker access to the private inbox of the attacker proxy app. It mimics an app which intentionally gives the attacker access to its private communications. Line 201 listens for messages of type appid on the pub channel. Since the appid of an app is private in the model, the only app whose id is broadcast is one defined to be malware (see lines 189–

190). Line 202 listens for a message y on the appid's private channel inbox(a). Line 203 publishes y on pub channel, giving it to the attacker.

```
186 free phoneUser:Principal.
187 let AttackerProxy() =
188 (!get apps(a) in
189       if Malware(a) then
190       out (pub,a))
191 | (!in (pub,a:appid);
192       get apps(=a) in
193       let r = mkResource(phoneUser,a) in
194       out (pub,r))
195 | (!in (pub,x:command);
196       out (filesystem,x))
197 | (!in (pub,x:command);
198       out (log,x))
199 | (!in (pub,x:command);
200       out (contentprovider,x))
201 | (!in (pub,a:appid);
202       in (inbox(a),y:bitstring);
203       out (pub,y)).
```

The following snippet defines functions and types which will can be used by applications for simple key definition and encryption with authentication. Line 204 defines a type Secret which will be used for secret objects. Line 205 defines the symkey type for symmetric keys . Line 206 defines a table which stores user information (Principal, see line 1 above) and an associated secret. Line 207 defines a key definition function kdf. It takes a Secret parameter and generates a symmetric key symkey based on it. Line 208 defines a function for performing authenticated encryption (authenc). It takes a symmetric key parameter and a resource parameter which is the contents to encrypt. Line 209 defines a destructor for encryption. Given the correct symkey and an encrypted resource, the authdec will return the original resource r.

```
204 type Secret.
205 type symkey.
206 table userCredentials(Principal,Secret).
207 fun kdf(Secret): symkey.
208 fun authenc(symkey,resource): resource.
209 reduc forall k:symkey,r:resource; authdec(k,authenc(k,r)) = r.
```

The following snippet defines and announces events in the model. *Events* are things which can be announced by functions to indicate that things of note have occurred. They can not be announced by the attacker and so can be used to trace the progress of the model. We use event announcements in queries to the ProVerif engine as shown below. Line 210 defines an event which is announced

when a resource has been leaked. The ResourceLeak event includes the appid of the owner of the resource and its resourceId. Line 211 defines an event announced when a file has been created in an app's private area. It includes the appid of the owner of the private folder, its resourceId, and the appid of the resource's owner (in case it is not the app storing it).

Lines 212–220 define a function SecurityGoals() which uses the events to make announcements. It has three subprocesses which listen for communication on channels.

Lines 213–214 define a process which listens for leaked resources. Line 213 listens for resource objects sent on the pub channel. Such a resource has been leaked to the attacker, so line 214 announces the ResourceLeak event for the r leaked.

Lines 215–217 define a process which examines the files table and finds files which have been created in an app's private space. Line 215 non-deterministically chooses an app a from the apps table. Line 216 non-deterministically retrieves one of a's files from files which is stored in the location private(a) (see line 37 above). Line 217 announces PrivateResource with r, the file's resource, the resource's owner id id(r), and the resource's owner owner(r).

Lines 218–220 define a similar process for web based private files. As in the previous process, line 218 non-deterministically chooses an app a, line 219 non-deterministically chooses a file on a's web location web(a) (see line 38 above), and announces it using PrivateResource.

```
210 event ResourceLeak(appid,resourceId).
211 event PrivateResource(appid,resourceId,appid).
212 let SecurityGoals() =
213 (!in (pub,r:resource);
214       event ResourceLeak(owner(r),id(r)))
215 | (!get apps(a) in
216       get files(=private(a),p,pp,f,fp,r) in
217       event PrivateResource(a,id(r),owner(r)))
218 | (!get apps(a) in
219       get files(=web(a),p,pp,f,fp,r) in
220       event PrivateResource(a,id(r),owner(r))).
```

The following snippet contains queries that ask ProVerif to investigate the security and privacy properties of the model and apps. The first query on line 221 is a reachability check to ensure that ResourceLeak events are possible. If an event of interest in not reachable, there is usually a mistake in the model. The following three queries check possible valid reasons why a resource leak might occur. The last query checks whether private resources can be created, also a reachability check.

Lines 222–223 ask ProVerif to prove that whenever a ResourceLeak event is announced where resource r owned by o is leaked, it can be shown that app a had acquired readAuthorized on r previously and a is malware. Thus, the leak

can attributed to a malware app being given legitimate access to r by an app according to one of the rules in lines 24, 26–27, or 30–31 above.

Line 224 checks another possible reason why a resource might be leaked: if the resource's owner is malware. Malware might intentionally leak its own resources to an attacker. Thus, the query asks ProVerif if it can be shown that the event occurred on a resource which is owned by an app which has been declared Malware.

Lines 225–226 check another possible reason why a resource might be leaked: if the user authorized a malware app to read the resource. The model must take into consideration the user's actions since they are another valid mechanism which can lead to the leaking of a resource. The query asks ProVerif to prove that a resource leak of r which is owned by o occurred after the principal phoneUser (see line 186) had granted read authorization on r to a malware app a using UserAuthorizedRead (see line 19 above).

If none of the above valid series of events occurred, we have found an attack on the authorization mechanism protecting r.

Line 227 checks that the event PrivateResource is reachable.

```
221 query a:appid,r:resourceId; event(ResourceLeak(a,r)).
222 query o:appid,a:appid,r:resourceId;
223     event(ResourceLeak(o,r)) ==> readAuthorized(a,r) && Malware(a).
224 query o:appid,a:appid,r:resourceId; event(ResourceLeak(o,r)) ==> Malware(o).
225 query o:appid,a:appid,r:resourceId;
226     event(ResourceLeak(o,r)) ==> userAuthorizedRead(phoneUser,a,r) && Malware(a).
227 query a:appid,r:resourceId,b:appid; event(PrivateResource(a,r,b)).
```

### A.2 App Models

We use the model code from the previous section to examine how apps behave in the Android environment. As shown in the app examples in section 3, other apps and features of the Android environment can open the door for attacks on even relatively secure apps. We therefore test the app models in the ProVerif model of the Android environment and ask the ProVerif engine to attempt to prove queries. In the following snippets we show models for five of the apps which we analyzed as part of this study: Google Drive, GMail, Yahoo! Mail, Dropbox, and BoxCryptor.

The following snippet sets up the communication channels which will be used by the app models which follow. Line 228 defines a channel openWith which can be used by the user to request that a file be opened with an app. The channel allows apps to delegate the opening of files to other apps. For example, GMail delegates the viewing of attachments to external viewer apps. Line 229 defines a appname object googledriveid which identifies the Google Drive appname. The

appname is defined as private, so the attacker does not know it. Lines 230–233 define similar appname objects for the GMail, Yahoo! Mail, Dropbox, and BoxCryptor apps respectively.

Lines 234–238 define functions which enable the models to reference the appid behind each appname (see line 4 above). Line 234 defines a function googledrive() to access the appid of the Google Drive app. The googledrive() function generates an appid for the private appname defined for the app on line 230. Lines 235–238 define similar functions to define the appids meant for the GMail, Yahoo! Mail, Dropbox, and BoxCryptor apps respectively.

```
228 free openWith: channel.
229 free googledriveid: appname [private].
230 free gmailid: appname [private].
231 free yahoomailid: appname [private].
232 free dropboxid:appname [private].
233 free boxcryptorid: appname [private].
234 letfun googledrive() = app(googledriveid).
235 letfun gmail() = app(gmailid).
236 letfun yahoomail() = app(yahoomailid).
237 letfun dropbox() = app(dropboxid).
238 letfun boxcryptor() = app(boxcryptorid).
```

The following snippet defines a function which models the behavior of the Google Drive app. Line 239 defines the function GoogleDrive(). It requires a Principal parameter which represents the user on whose behalf the function runs (see line 1 above). The function contains three subprocesses which perform tasks for the app. The first subprocess runs only once; the second and third run under unbounded replication.

Lines 240–241 define a process which define the app and its locations. Line 240 inserts the googledrive() appid into the apps table, thus registering it as an app. As shown in the previous section, many processes in the Android model use the apps table to determine which apps are installed on the phone (*ex.* FileSystem() on line 125). Line 241 publishes Google Drive's private storage location name private(googledrive()) and web site web(googledrive()) on the pub channel, thus informing the attacker of them. The web site and root directory of an app are not secret, so we ensure that the attacker is aware of them, even if it is not aware of the directory structure beneath them.

Lines 242–245 define a process which creates new files for the Google Drive app on its web site. Line 242 creates a new file name f. Line 243 creates a new path p. Line 244 creates a new resource r on behalf of the user u and the googledrive() appid. Line 245 inserts the new file in the files table on the Google Drive web site web(googledrive()) with path p, file name f, and resource contents r. It is defined with no path or file permissions (noPerms()).

Lines 246–254 define a process which opens a file for the Google Drive app by downloading a file from the web site to its private storage and opening it with another app. This models the behavior of the Google Drive app since it stores

official copies of its files on the Google Drive web site and downloads them to private storage for on-phone reading or editing. Line 246 receives a message on the openWith channel, modeling a situation where the user chooses to open a file via the Google Drive app (see line 228 above). Line 247 non-deterministically chooses a file from Google Drive's web site web(googledrive()) via the files table. The file's path p, path permissions pp, file name f, file permissions fp, and resource contents r are selected from the table. Line 248 non-deterministically chooses another app a from the apps table to send the file to. Line 249 checks that the user phoneUser has authorized the intended recipient app a to read the resource id of the file id(r). As discussed above on line 19, no clause leads to UserAuthorizedRead, but since it is declared as block, ProVerif will attempt to assume it to discover what outcomes follow from it. Thus, ProVerif may attempt to assume UserAuthorizedRead for a and enable the process to proceed. As shown above on lines 226–227, the user giving read authorization to a malware app is an acceptable way for a file to be leaked. Line 250 creates a new path p1 for the file. Line 251 defines new path permissions pp1 with world executable (traversable) permission set. Line 252 defines new file permission fp1 with world readable permission set. Line 253 makes a copy of the file by inserting a new file in the files table located in Google Drive's private location private(googledrive()) at the new path p1, with new path permissions pp1, the old file name f, new file permissions fp1, and the old resource object r. Line 254 then sends the path and file name information to the recipient app a via the explicitintent channel (see line 116 above). The explicit intent contains the new path name p1 and file name f. Because the path is world executable and the file is world readable, the recipient a will be able to read the file (see lines 95–97 above).

```
239 let GoogleDrive(u:Principal) =
240     (insert apps(googledrive());
241     out (pub, (private(googledrive()),web(googledrive()))))
242 | (!new f:filename;
243     new p:path;
244     let r = mkResource(u,googledrive()) in
245     insert files(web(googledrive()),p,noPerms(),f,noPerms(),r))
246 | (!in (openWith,());
247     get files(=web(googledrive()),p,pp,f,fp,r) in
248     get apps(a) in
249     if userAuthorizedRead(phoneUser,a,id(r)) then
250     new p1:path;
251     let pp1 = setExecutable(noPerms()) in
252     let fp1 = setReadable(noPerms()) in
253     insert files(private(googledrive()),p1,pp1,f,fp1,r);
254     out (explicitintent(googledrive(),a),(p1,f))).
```

The following snippet contains a function which models the behavior of the Gmail app. Line 255 defines a function called Gmail(). Like the previous model, it requires a Principal parameter which represents the user on whose behalf the

process runs (see line 1 above). It too has three subprocesses which perform tasks for the app. The first subprocess runs only once; the second and third run under unbounded replication. Their tasks are similar to the ones in the Google Drive app.

Lines 256–257 define a process which defines the app and its locations. Its functions parallel the functions of lines 240–241 above. Line 256 inserts gmail() into the apps table. Line 257 publishes the location of its private directory and web site on the pub channel, thus disclosing them to the attacker.

Lines 258–261 define a process which creates new files for the Gmail app on its web site. It functionality is similar to lines 242–245 above. Line 258 creates a new file name f. Line 259 creates a new path p. Line 260 creates a new resource r. Line 261 adds a new file to the Gmail web site web(gmail() with file name f, resource r, path p, and no file or path permissions.

Lines 262–269 define a process which allows attachments to be opened via the Gmail app by downloading the attachment to the phone and sending a content URI link to another app to open it. This models the behavior of the Gmail app since it downloads files to its private storage space and allows other apps to access them via a protected content provider. Line 262 receives a request on the openWith channel to open a file via the Gmail app (see line 228 above). Line 263 non-deterministically chooses a file from Gmail's web site web(gmail()) via the files table. The file's path p, path permissions pp, file name f, file permissions fp, and resource r are selected from the table. Line 264 non-deterministically selects another app a from the apps table to open the file with. Line 265 checks that the user (phoneUser) has authorized a to open the file. As noted in the previous model, userAuthorizedRead is a block which will be assumed by ProVerif as part of its processing. Line 266 creates a new path p1 to store the file. Line 267 creates a new contenturi called u with Gmail's private storage private(gmail) as the domain and p1 as the URI's path (see line 41). Line 268 registers u in the content table, adding the contenturi, the resource r that it refers to, and adding a to the list of allowed readers. The list of allowed writers is empty (nil()). Line 269 sends an explicit intent from the Gmail app to the recipient a with the new u as the data.

```
255 let Gmail(user:Principal) =
256     (insert apps(gmail());
257     out (pub, (private(gmail()),web(gmail()))))
258 | (!new f:filename;
259     new p:path;
260     let r = mkResource(phoneUser,gmail()) in
261     insert files(web(gmail()),p,noPerms(),f,noPerms(),r))
262 | (!in (openWith,());
263     get files(=web(gmail()),p,pp,f,fp,r) in
264     get apps(a) in
265     if userAuthorizedRead(phoneUser,a,id(r)) then
266     new p1:path;
267     let u = contenturi(private(gmail()),p1) in
```

```
268       insert content(u,r,cons(a,nil()),nil());
269       out (explicitintent(gmail(),a),u)).
```

The following snippet defines a function which models the behavior of the Yahoo! Mail app. Line 270 defines a function called YahooMail(). Like the previous two functions, it requires a Principal parameter. It too has three subprocesses which perform tasks for the app. The first subprocess runs only once; the second and third run under unbounded replication. Their tasks are similar to the ones in the Google Drive and Gmail apps.

Lines 271–272 define a process which defines the app and its locations. Line 271 adds the yahoomail() app to the apps table. Line 272 publishes the name of Yahoo!'s private directory private(yahoomail()) and its web site web(yahoomail()) to the attacker.

Lines 273–276 define a process which creates new files for the Yahoo! Mail app on its web site. It is similar to the behavior of the parallel process is Google Drive (lines 242–245) and Gmail (258–261) apps.

Lines 277–285 define a process which allows attachments to be opened via the Yahoo! Mail app by downloading the attachment to the phone and sending a content URI link to another app to open it. Much of its functionality is similar to the behavior of the open with process in the Gmail app (262–269). As noted in section 3, Yahoo! Mail's app stores attachments on the SD Card, so the model behaves accordingly. Line 277 receives a message to open a file attachment on the openWith channel. Line 278 non-deterministically chooses a file from the files table. Line 279 non-deterministically chooses an app a to open the file with. Line 280 checks that the user has authorized a to open and read the file. Line 281 makes a copy of the file chosen on the SD Card by making a new entry in the files table in the sdcard() location with the path p, file name f, and resource r from the web. Since the file is written to the SD Card, the comment notes that it has been effectively leaked to the attacker. Line 282 creates a new path p1 which is used in line 283 to create a contenturi for the attachment. Line 284 adds the new uri ur to the content table so it can be resolved by the content provider process (see lines 166–177 above) and adds a to the list of authorized readers. Line 285 sends the new ur to a via an explicit intent.

```
270 let YahooMail(u:Principal) =
271       (insert apps(yahoomail());
272       out (pub, (private(yahoomail()),web(yahoomail()))))
273 | (!new f:filename;
274       new p:path;
275       let r = mkResource(u,yahoomail()) in
276       insert files(web(yahoomail()),p,noPerms(),f,noPerms(),r))
277 | (!in (openWith,());
278       get files(=web(yahoomail()),p,pp,f,fp,r) in
279       get apps(a) in
280       if userAuthorizedRead(phoneUser,a,id(r)) then
281       insert files(sdcard(),p,noPerms(),f,noPerms(),r); (* This line leaks the resource *)
```

```
282     new p1:path;
283     let ur = contenturi(private(yahoomail()),p1) in
284     insert content(ur,r,cons(a,nil()),nil());
285     out (explicitintent(yahoomail(),a),ur)).
```

The following snippet contains a function which models the behavior of the Dropbox app. Line 286 defines a function called Dropbox(). Its structure is very similar to the structure of the Yahoo! Mail app discussed previously.

Lines 287–288 define a process which defines the app and its locations. Its code is identical to the first subprocess of the Yahoo! Mail app (lines 271–272 above).

Lines 289–292 define a process which creates new files on the app's web site. Its code is identical to the second subprocess of the Yahoo! Mail app (lines 273–276 above).

Lines 293–298 define a process which allows files on the Dropbox web site to be opened via the Dropbox app. The model shows the behavior described above in section 3: downloading the file to the phone, storing it on the SD Card in a cache directory, and sending a link to the file to the opening app. Line 293 receives a message to open a file which is stored on the Dropbox web site. Line 294 non-deterministically chooses a file from the files table which is stored on the web site (web(dropbox())). Line 295 non-deterministically selects an app to open the file. Line 296 checks that the user has authorized a to read the file. Line 297 makes a copy of the file on the SD Card by creating a new entry in the files table in the sdcard() location with the path p, file name p, and resource r retrieved from the web site. It is stored with no file or path permissions. As noted by the comment, once the file has been stored in the sdcard() location, it is readable by the attacker. Line 298 sends a link to the file to a via an explicit intent. The intent includes the path and file name of the file to open (p,f).

```
286 let Dropbox(u:Principal) =
287     (insert apps(dropbox());
288     out (pub, (private(dropbox()),web(dropbox()))))
289 | (!new f:filename;
290     new p:path;
291     let r = mkResource(u,dropbox()) in
292     insert files(web(dropbox()),p,noPerms(),f,noPerms(),r))
293 | (!in (openWith,());
294     get files(=web(dropbox()),p,pp,f,fp,r) in
295     get apps(a) in
296     if userAuthorizedRead(phoneUser,a,id(r)) then
297     insert files(sdcard(),p,noPerms(),f,noPerms(),r); (* This line leaks the resource *)
298     out (explicitintent(dropbox(),a),(p,f))).
```

The following snippet defines a function which models the behavior of the BoxCryptor app. Line 299 defines a function called BoxCryptor(). Its structure has similarities to the previously defined apps, but there are significant differences because BoxCryptor, an encryption layer on top of commercial cloud file storage services, uses another app's storage area to keep its encrypted files.

Lines 300–301 define a process which defines the app and its locations. Its code is identical to the first subprocess of the Yahoo! Mail app (lines 271–272).

Line 302–303 define the Dropbox app. The model does this because BoxCryptor is an app which uses other apps' storage. In this case it is shown to store all of its files in the Dropbox app's web site and storage area. Therefore, line 302 inserts dropbox into the recognized apps table in case the model for Dropbox is not already defined. Line 303 then publishes the location names to the attacker.

Lines 304–309 define a process which creates new files for the BoxCryptor app. Since BoxCryptor uses Dropbox's storage area, all files are written to web(dropbox()). Line 304 chooses a new file name f. Line 305 defines a new path p. Line 306 creates a new resource for the file, defining u as the principal which created it and dropbox() as the owner app. Line 307 gets the user's BoxCryptor password pwd from the userCredentials table (see line 206 above). Line 308 encrypts the contents of the resource r using the authenc encryption function (see line 208 above). The key for the encryption is derived from pwd using the key derivation function kdf (see line 207 above). The encrypted contents of the file is er which is also a resource. Line 309 creates a new file on Dropbox's web site web(dropbox()) with the given path, file name, and encrypted contents.

Lines 310–321 define a process which allows the opening of files encrypted with BoxCryptor. Line 310 listens for messages on the openWith channel. Line 311 non-deterministically chooses a file from the Dropbox web site. Line 312 non-deterministically chooses an app from the apps table to open the file. Line 313 gets the user's password pwd from the table of user credentials. Line 314 decrypts the encrypted contents of the file by deriving the encryption key using the key derivation function kdf on pwd. The decrypted contents dr are also a resource. Lines 315–316 check that the user has authorized read and write permission to the app a which will open the file. Assuming the user has, line 317 creates a new path dp to store the decrypted file in the plain text file cache and line 318 creates a new path ep to store the encrypted file in the encrypted file cache. Line 319 stores an encrypted copy of the file in the encrypted file cache on the SD Card. Line 320 stores a decrypted copy of the file in the decrypted file cache on the SD Card. Line 321 sends an explicit intent to a with the path to the decrypted file in the decrypted file cache. The intent is sent as if from dropbox() since it is the app which actually stores the files and appears to be in charge of them.

```
299 let BoxCryptor(u:Principal) =
300       (insert apps(boxcryptor());
301       out (pub, (private(boxcryptor()),web(boxcryptor()))))
302 | (insert apps(dropbox());
303       out (pub, (private(dropbox()),web(dropbox()))))
304 | (!new f:filename;
```

```
305       new p:path;
306       let r = mkResource(u,dropbox()) in
307       get userCredentials(=u,pwd) in
308       let er = authenc(kdf(pwd),r) in
309       insert files(web(dropbox()),p,noPerms(),f,noPerms(),er))
310   | (!in (openWith,());
311       get files(=web(dropbox()),p,pp,f,fp,r) in
312       get apps(a) in
313       get userCredentials(=u,pwd) in
314       let dr = authdec(kdf(pwd),r) in
315       if userAuthorizedRead(phoneUser,a,id(dr)) then
316       if userAuthorizedWrite(phoneUser,a,id(dr)) then
317       new dp:path;
318       new ep:path;
319       insert files(sdcard(),dp,noPerms(),f,noPerms(),r); (* Encrypted file in cache *)
320       insert files(sdcard(),ep,noPerms(),f,noPerms(),dr); (* Decrypted file in cache*)
321       out (explicitintent(dropbox(),a),(dp,f))).
```

The following snippet defines the process which ProVerif uses to perform the queries on the models. It is composed of the functions which been defined above. Here the user can select which of the apps to query in a given experiment. Line 322 defines that the functions inside the parentheses will be run by the ProVerif simulator as part of its active process. Line 323 initializes the AppInstaller() function for installing apps with the approval of the phoneUser, the FileSystem() function for managing the files on the phone, the Log() function for logging events, the Binder() function to enable intent routing, and the ContentProvider() function to enable content URI resolution. The functions are separated by the | symbol to indicate that they are to run in parallel. Line 324 initializes the AttackerProxy() function to enable the attacker to access phone services and the SecurityGoals() function which defines events for security properties that we want ProVerif to check. Lines 325–329 initialize the functions for the apps defined above. As shown, each app is initialized with the phoneUser.

Note that in the snippet below, all of the app functions are run in parallel. If the user only wants ProVerif to test a selection of them, the others can be commented by placing them between (∗ and ∗).

```
322 process (
323       AppInstaller(phoneUser) | FileSystem() | Log() | Binder() | ContentProvider()
324       | AttackerProxy() | SecurityGoals()
325       | Dropbox(phoneUser)
326       | BoxCryptor(phoneUser)
327       | GoogleDrive(phoneUser)
328       | Gmail(phoneUser)
329       | YahooMail(phoneUser)
330 )
```