Authorization Logics For Mobile Ecosystems

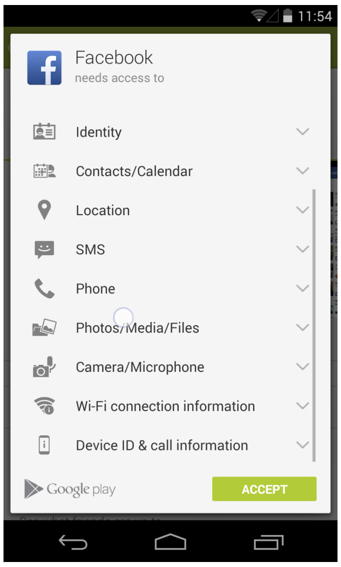
Thesis Proposal

Joseph Hallett

July 4, 2014

Contents

List of Figures


11:54

Facebook
needs access to

Identity
Contacts/Calendar
Location
SMS
Phone
Photos/Media/Files
Camera/Microphone
Wi-Fi connection information
Device ID & call information

Figure 1: Permissions requested by the Facebook app on Android. When installing an app a user is presented with a list of permissions the app requires to run. Permissions describe what phone features an app will have access to.

1. Introduction

Android is an operating system for mobile phones. Users run prepackaged software (*apps*) which they download from special app stores.

When an app is installed on Android the user is prompted to accept the privileges required by the app. The user makes a decision based on what they know about the app and their own personal security policies. Most users accept the app without thinking about it [32]. They do this for a many reasons: because they don't understand the risks, they don't understand the permissions, or they simply don't care and will install the app whatever.

Facebook is an example of an app which requests many permissions (Figure 1). Users trust Facebook not to be malicious even though it has access to amount of their personal data. Some apps are over privileged [31]: they request permissions that grants them access to data they do not use. Some apps are malicious [68]: they request permissions to steal data or to spend money without the users consent. Other apps are *potentially unwanted software (PUS)*[1]: these are apps which are generally not malicious but may have features that goes against what the user wants. On a PC this might be a browser tool bar bundled with another program. On Android PUS might be an aggressive advertising framework that leaks private information or repeatedly polls GPS information, draining the phones

[1]Whilst Google favor the term PUS to describe this kind of malware other names are also used. PUA (potentially unwanted applications) and PUP (potentially unwanted programs) are other names for the same family of malware. These terms can be used interchangeably.

battery [59].

More generally users and computers make decisions. Whether it is to update an app or to connect to a website: the decisions are made based on the security policies and trust relationships of the user and device. These security policies may include the use of tools or experts to decide whether something is malicious. For instance a user may trust a firewall program to enforce their network policy; and they may trust a tool like Shorewall [27, 60] to write policy for them. Alternately a user might wish to be able to install apps but only trust apps Amazon have vetted to be installed on their device. *Broadly, the aim of this research is to formalize these security policies so they can be studied precisely and enforced automatically.*

Mobile operating systems are similar to existing systems but have a different trust model and are used differently. Software is bought and downloaded from app stores, Apps run within sandboxes and collaborate to share data. The devices contain more personal data than before: sensors tracking users' locations, gyroscopes measuring how users move, and microphones listening to users calls. The bring your own device (BYOD) trend encourages users to take the devices they have at home into work. This creates a tension between how the corporate IT department may require employees to use their devices and the user's policies on how they want to use their devices. These features add a novel challenge to modelling these devices and the stores and users surrounding them.

Formalizing policies allows comparisons to be made between different systems and the user's policies. Common comparisons the two biggest mobile OSs, iOS and Android, are informal: iOS is closed, more of a *walled garden*[8, 5]. Apps go through a vigorous review process and Apple is selective about what it sells. Android is more permissive. With a formal language to describe system policy we can make a precise comparison.

To analyze permissions and detect malware both static and dynamic analysis has been used. Static analysis tools infer (and occasionally enforce) complex security properties about the code. What is missing is the link between the assurances these tools can give and the *user-level policies* we want to enforce. A user-level policy describes how a user wishes an app to behave; though a user may only specify them informally.

By using an *authorization logic* as the glue layer we can enforce the policy by building on the work on access control in distributed systems. Static analysis tools can be trusted to give statements about code, as can other analysts and principals, that can be combined to implement a security policy.

This thesis research will show how authorization logics can be used to make security decisions in mobile devices. Security decisions are made manually by smart phone users and it is our belief that by automating these choices users can avoid having to make security decisions and their overall security be improved. To do this we plan:

- *To model the decisions and trust relationships inherent in Android and other mobile operating systems.* We will write security policies that describes the current state in these systems and serve as a base to compare systems.

- *To instantiate a logic of authorization that allows us to model the trust relationships between the components of an operating system and the users.* This will include using static (and dynamic) analysis tools to make decisions. These tools will be introduced as *principals*: entities which say things about others. The logic will be able to model what happens when apps can collude. The logic will be based on earlier work on the *SecPAL language* [12] that has been used for distributed access control decisions.

- *To implement an app store that serves users only the apps that meet their security policies.* This will include a user-study where we evaluate how well users comprehend their policies and the decisions made for them. This may lead into generating proof-carrying code certificates [50] for apps that allow a device to check that their policy was met without having to do the full inference themselves.

- *To study how users understand their security policies and the ways these policies are enforced.* SecPAL is claimed to be more readable compared to other authorization logics and access control languages[38]. Whilst end-users may not want to write their own policies system administrators and expert users should be able to comprehend what a policy means; they should understand why their policy allows some decisions and not others.

- *To explore how security policies change with time and when apps can collude.* A user's security policy need not be static. People change jobs and may bring old devices to new environments requiring new security policies. Apps can collude: two apps might meet a security policy when considered on their own but together they might act to share data inappropriately. Over time an app might want greater access and increased permissions to support new functionality. If this increased functionality breaks policy what should happen? What should happen when a policy changes on a device or is revoked entirely? It is not obvious how to write and check security policies for these scenarios; or how to enforce the policy at runtime.

1.1. A Logic of Authorization For Mobile Devices

Logics of authorization are used to decide whether some system may do something. This might be carrying out an action, accessing data, or describing what another entity can do. When we apply these logics to files and information we create an access control system. A review of the history and applications of the logics is given later in this proposal (page **??**); but in summary they have shown themselves to be useful for modelling the complex security and trust policies in modern systems.

Mobile devices are different to traditional computers. They have more information about their users. They don't offer the user the traditional file system interfaces. Everything is sandboxed and closed software markets (app stores) distribute software. The app stores typically allow developers to sell their apps but are selective about what they will sell. Apps are vetted for quality and security[2]. Static and dynamic analysis tools are

[2]We believe: very few stores document their policies. The *Firefox Marketplace* is a notable exception

used as well as traditional inspection. The policies the app stores apply to their apps form an authorization decision (*the analysis team says app can be sold*) and there is a delegation of trust to the analysts and their tools. It is not clear how these policies and trust relations filter through to the end users.

These differences amount to a different model of trust than traditional machines such as PCs and embedded systems. There are a new set of authorization problems that are not obvious how to express in some authorization logics. One problem is how an app store should convince a device an app meets its policy; if checking a policy cannot be done on a phone (maybe the battery is running low) can checking be delayed? What happens when two trusted principals disagree (as might happen with apps in different stores)? Some languages, like Cassandra [11], authorize on the basis of the speaker holding a role; but what are the roles when a store has a changing policy?

To solve these problems we have taken an existing authorization logic, SecPAL [12], and extended it with a series of predicates that can describe how security policies are met inside a mobile ecosystem. Future work will include describing the current security policies for Android and other mobile OSs as well as the app acceptance policy for some app stores. This will allow comparisons to be made between them, and (with a database of apps) comparing what kinds of apps different markets allow.

1.2. Compound Policies Over Time

Consider a user who has a smart phone and is buying apps. The user must decide if they want to install an app: to do this they apply a series of judgements which we may informally call their personal *security policy.*

The user has their own security policy. They also have other security policies they implicitly follow. When they download apps from an app store they also gain the security policy of the store and what it will sell. If the phone runs in a corporate environment then they may also be subject to the company's corporate policy. The operating system itself may have certain restrictions on what it will allow. The APK app format used on Android can also be installed on Blackberry and Sailfish operating systems. Each system may add additional restrictions that may make some apps not installable. Sometimes these policies need to be combined to create a policy that will work in multiple environments and on a system with its own rules. An example of how a compound policy might be written is shown in Figure 3.

Suppose the phone uses this policy for a while but the user changes jobs. Now they have to meet a new `ITDeptPolicy` set by a different administrator. Should any installed apps be uninstalled if they don't meet the new policy? If we already have a certificate showing the apps passed the old policy can we reuse it to create a new certificate that shows the app meets any additional restrictions?

as they publish their review criteria online: `https://developer.mozilla.org/en-US/Marketplace/Submission/Marketplace_review_criteria`. Apple do publish a long list of guidelines as to what will be accepted or not, but it is not exhaustive and does not state how they check: `https://developer.apple.com/appstore/resources/approval/guidelines.html`.

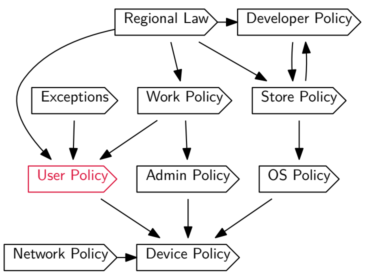


Figure 2: Compound policies applied to one another. Users, businesses and stores are all subject to regional laws. A store may have a policy but the developers who write the apps may also add their own rules in. Devices might have their own policies set by their designers but also have the OS policies. If the device is used on a network certain traffic may be restricted. The user's policy (in red) is just one component of this ecosystem and cannot be considered on its own.

What about the data an old version of an app may have stored? If an app were to reduce its permissions but still have access to the data then there is a risk of an information leak. Simply deleting the old data isn't good enough as a user may still need their documents.

Whilst other authorization logics have looked at making one-time decisions about whether to allow a computer to make a decision; there has been less work on modelling these policies over time and seeing how a changing security policy affects a changing device.

Alternatively say there is an app which the developer is continually improving and adding new features. When the app is installed it may meet the security policy but with increasing features requiring access to more permissions and introducing more complexity or a change of advert library the app no longer meets the security policy.

Should the app be removed? If the app is used every day by then the user may not be pleased that the phone has decided to break their favorite app; regardless of whether the fault lies with app developer or the policy designer. Equally just stopping updates for the app increases app version fragmentation and reduces security by rejecting bug fixes. Allowing the update isn't correct either as it means breaking the security policy.

Whilst there have been several papers looking at (and proposing methods to stop) excessive permissions in applications [31, 64] there has not been a thorough review of how permissions change for apps over time and between versions of the same app as far as we know.

```
Phone says app is-installable
  if app meets UserSecurityPolicy,
     app meets AppStorePolicy,
     app meets ITDeptPolicy,
     app meets OSPolicy.

Phone says User can-say∞
  app meets UserSecurityPolicy.

Phone says PlayStore can-say0
  app meets AppStorePolicy.

Phone says ITAdmin can-say∞
  app meets ITDeptPolicy.
```

Figure 3: A compound security policy where an installation policy for a phone is dependent on other security policies.

1.3. Curated App Stores

Apps are normally distributed on mobile devices through an app store. On iOS users have the *App Store*: a curated market place run by Apple (though other, albeit clunkier, distribution mechanisms do exist such as the *over the air (OTA)* update mechanism used for testing and some apps banned from the App Store[3]) that is perceived as being picky about the apps it sells.

Android users have a far greater choice of marketplace. The *Play Store* is the app store distributed by Google. It is less moderated than Apple's store. Amazon have their own app store that serves as a more curated version of Google's offering. It is the default on their Kindle tablets. Other app stores target specific regions: such as *Anzhi* and *gFan* in China, or the *SK T-Store* in Korea. Some, such as *Yandex.Store, AppsLib and SlideMe*, are pre-installed by OEMS who can't or don't want to meet Google's requirements for the PlayStore. The *F-Droid* store only delivers open source apps. Others exist to distribute pirated apps.

On average eight percent [4] of the apps in each of these alternative market places is malware. The Play Store contains very little malware however (0.1% of total apps), whilst a third of the app in the Android159 store were found to be malicious.

Every app store has a different security policy. They enforce these policies when they pick which apps to sell to their users. By using an authorization logic to decide whether apps will meet a security policy we have the ability to create a new kind of app store where offerings are tailored to the user's security policy. By creating app stores tailored

[3]An example of this would be the *GBA4iOS* emulator: (`http://gba4ios.angelxwind.net/download/`. Emulator apps are seen to support video game piracy so Apple does not allow them to be sold in the App Store.

Store	Region	Apps	Downloads (per month)	Security	Notes
PlayStore	All	800×10^6	2.5×10^9	Estimated 0.01% malware (F-Secure labs)	The default app store for Android devices.
Yandex.Store	Russia	50×10^3		Anti-virus scanning provided by Kaspersky.	Pre-installed by six OEMs. Used as the Android-app app store on the Jolla operating system.
Anzhi	China	180×10^6	2.2×10^3	Estimated 5% malware (F-Secure labs)	Quarter of a million users.
SK-T Store	Korea	70×10^6	28×10^6		
SlideME	All	40×10^3	15×10^3	Using multiple malware scanners including one by *BlueBox security* that can detect apps exploiting the master key vulnerability.	Installed by 140 OEMs. Twenty million users.
Amazon AppStore	All	76×10^3	25×10^6		Used on Kindle tablets, but popular on Android.

Figure 4: Summary of different app stores for Android (data taken from the *One Platform Foundation* list of App Stores: `http://www.onepf.org/appstores/`).

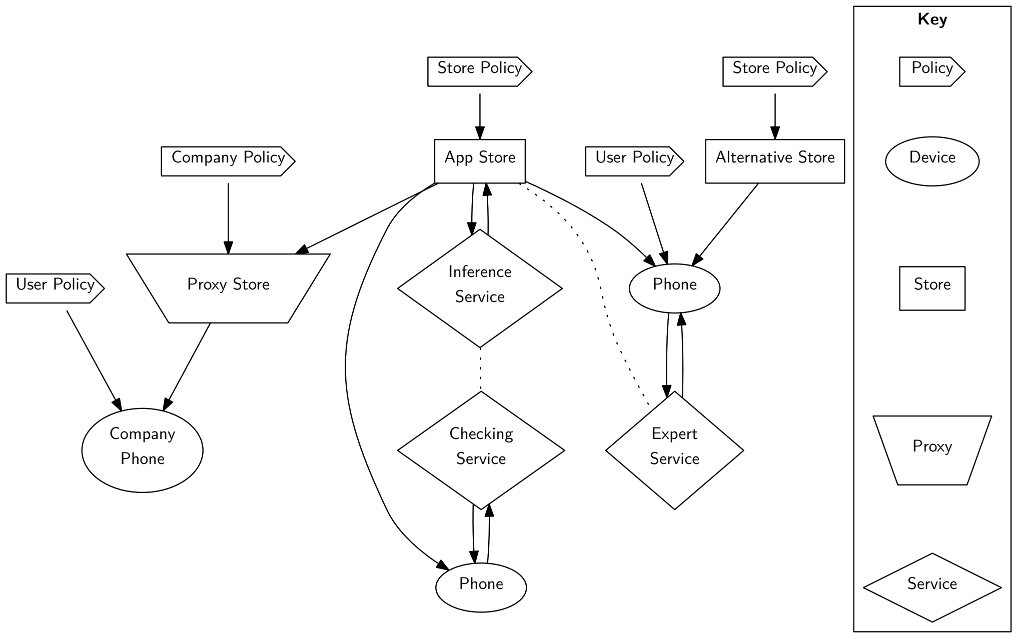


Figure 5: Overview of the entities in App Guarden.

to a security policy we also give ourselves a way to empirically measure how restrictive a security policy is: we can measure the number of apps offered inside the stores.

1.4. Project Context (App Guarden)

This thesis will form part of the *App Guarden*[4] project. The App Guarden project aims to improve the quality of mobile security by developing new tools to analyze apps and the app stores that sell them.

This work contributes by developing the security policies that describe what the user wants and showing how they can be enforced using the tools that can check security policies within the code. The end result might be a system as shown in Figure 5 where devices are interacting with stores and security services such as static analysis tools or proof checkers.

Between each of the nodes different policies could be enforced. Consider an *app store* and *proxy store*. There is a master app store that sells many apps. A company provides its employees each with a phone that they can install apps on but they have their own security and usage policy set by their IT department. Employees shouldn't install anything that breaks the policy. On all the devices the IT department provide they

[4]`http://groups.inf.ed.ac.uk/security/appguarden/Overview.html`

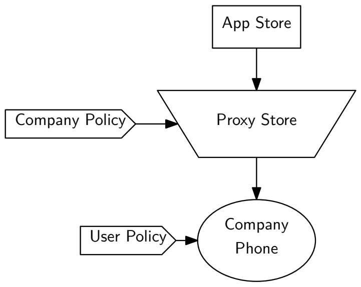


Figure 6: Security policies and the proxying store

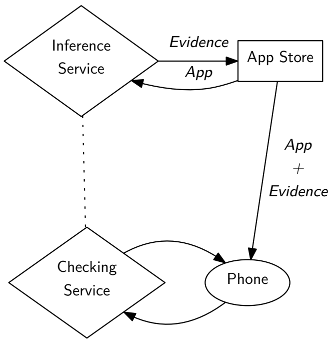


Figure 7: Checking services and an app store.

install a special proxy app store. The proxy app store takes apps from the main store but discards any apps which might break the policy it is supplied with. Users may download apps from the company store, but they might exercise their own judgement and only download apps that meet their own policies. At each stage (as shown in Figure 6) judgements are being made about what is acceptable from an app store, and the policies are refined.

Proof-carrying authentication [2] and authorization logics such as BLF [65] have already introduced ideas from proof-carrying code into authorization logics. The focus of their work has been on access control where a user is providing a proof that they have the credentials to access a resource. In the scenario we propose the role of the user is reversed: the store offers many proofs to the user to increase their trust in its wares; rather than the user offering one specific proof to prove they have the right to complete a certain action. These proofs also form an audit trail we can present to the user to say why a decision was made. This provides a valuable justification for each decision, as well

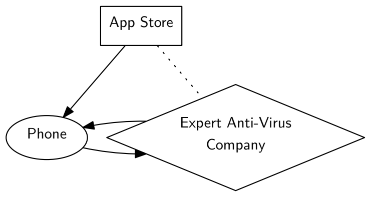


Figure 8: Use of an expert checker.

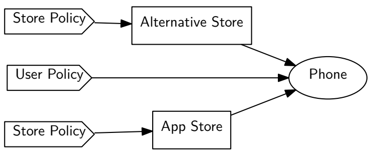


Figure 9: A device using multiple stores with different policies.

as aiding the debugging of policies.

The app store sells apps and it wants to reassure its users that it guarantees that the app it sells meets the policies it claims. Being able to infer these properties is complex and takes both time and battery power; this is difficult as many phones are battery constrained. To avoid this the app store uses an inference service to produce digital evidence to be supplied with the app that shows (with the aid of a checking service that could be running on the device) that an app meets the policy (as shown in Figure 7). Other parts of the App Guarden project are developing these tools.

An alternative form of this could be where a store delegates to an expert third party to make statements about the apps it sells. One might imagine a scenario where an app store might claim *"We don't sell viruses in our store, but don't take our word for it: here's a well known anti-virus company that will verify our claim"* (as shown in Figure 8).

If a user is using multiple stores (for example a jail broken iPhone user might buy apps from both the App Store and the Cydia Store) then the policies the user might be applying become complex (as described in Figure 9). This leads to interesting questions around how policies should be composed and the equivalence of security policies when they are; as well as questions about the overall device policy in a system.

The work for this thesis will not concern itself with the development of tools to check the apps perform as they should; rather it will focus on modeling the trust relationships between these tools and the other entities in a mobile environment. This allows this part of the work to focus on the relationships and device policies rather than the intricacies of code analysis.

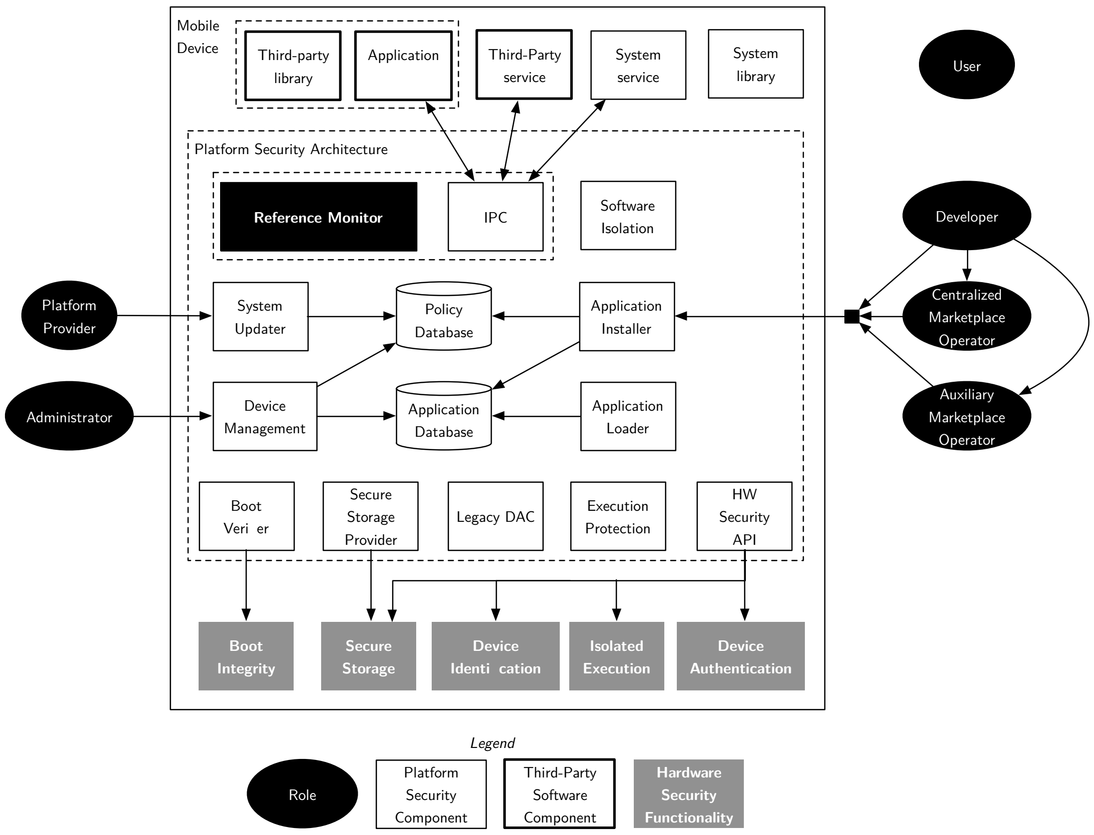


Figure 10: Overview of components in a mobile software platform like Android. Diagram taken from the *Mobile Platform Security* course at the University of Helsinki.

2. Review of Android Security

Since Android is used as the basis for many mobile OSs a review is given of its security features.

Android is a Linux OS for mobile phones and consumer electronics. It has a large software market of apps. Apps on the Dalvik virtual machine. Dalvik is a modified JVM architecture: it uses registers rather than stacks to save memory and reduce code size; and drops some type information (again for space). Apps use a sandbox provided by the OS that is based on Linux's permissions model [26].

2.1. Permissions and Apps

Android permissions are the access control mechanism users see most often. This section reviews their mechanisms and tools for analysing their use in apps.

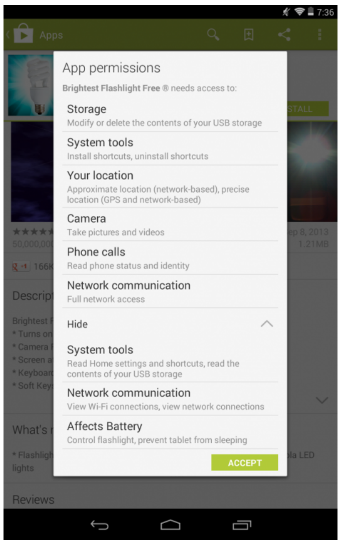
Android permissions come in three varieties: API permissions, file system permissions, and IPC permissions. API permissions say what high level functionality an app may

API Permission	File System Permissions
BLUETOOTH_ADMIN	net_bt_admin
BLUETOOTH	net_bt
BLUETOOTH_STACK	net_bt_stack
NET_TUNNELING	vpn
INTERNET	inet
READ_LOGS	log
READ_EXTERNAL_STORAGE	sdcard_r
WRITE_EXTERNAL_STORAGE	sdcard_r sdcard_rw
ACCESS_ALL_EXTERNAL_STORAGE	sdcard_r sdcard_rw sdcard_all
WRITE_MEDIA_STORAGE	media_rw
ACCESS_MTP	mtp
NET_ADMIN	net_admin
ACCESS_CACHE_FILESYSTEM	cache
DIAGNOSTIC	input diag
READ_NETWORK_USAGE_HISTORY	net_bw_stats
MODIFY_NETWORK_ACCOUNTING	net_bw_acct
LOOP_RADIO	loop_radio

Figure 11: Mappings between API and file system permissions on Android 4.4

access. For example the `INTERNET` permission allows apps to access the network. To enforce this Android uses the Linux file system permissions in the operating system: the `/etc/permissions/platform.xml` file defines mapping between the API and file system permissions. In this case any process started from an app with the `INTERNET` is assigned the `inet` file system permission which is used by the kernel to control access to network sockets. Not all API permissions are enforced through file system permissions: those which do are shown in Figure 11. Other API permissions are enforced through checks in code.

Every app is assigned a new unique file system permission at install time: creating the sandboxes apps run in. Apps with different file system permissions cannot access other apps data. A developer can request two apps run with the same permission by signing both with the same key. This is discouraged by Google as it can make collusion attacks easier. The Android Open Source Project (AOSP) (an open source version of Android used to port Android to different architectures) provides an example signing key for the system binary; some manufacturers do not change this key. If a rogue developer signs their app with this key they can escalate their privileges without declaring any


App permissions

Brightest Flashlight Free ® needs access to:

Storage
Modify or delete the contents of your USB storage

System tools
Install shortcuts, uninstall shortcuts

Your location
Approximate location (network-based), precise location (GPS and network-based)

Camera
Take pictures and videos

Phone calls
Read phone status and identity

Network communication
Full network access

Hide

System tools
Read Home settings and shortcuts, read the contents of your USB storage

Network communication
View Wi-Fi connections, view network connections

Affects Battery
Control flashlight, prevent tablet from sleeping

Figure 12: The *Brightest Flashlight Free* app prompting for its permissions at install time. This app is over privileged as a flashlight app should have no need for GPS or phone data, or network access. This extra functionality was used maliciously.

permissions [67].

IPC permissions are used when apps communicate with each other. Apps say what IPC messages (called *intents*) they will handle Intent filters allow these to be restricted to only apps with certain intent permissions and, if required, apps signed by the same key.

Apps must request API permissions at install time. The permissions are shown to the user: if the user disagrees with the permissions it cannot be installed. Often users do not look at these permissions; they accept them whatever is asked for [32]. This has led to malware and PUS that asks for too many permissions. This lets bad apps send premium text messages (a common monetization strategy [22]) or steal private information. Even if an app were to do nothing bad itself in itself; if the functionality exposed by the permission is exposed in an API then other apps could collude with the overprivileged app to gain privileges: as in the collusion attacks.

Tools can detect when an app is over privileged (like the app in Figure 12). The *Stowaway* tool [31] mapped Android permissions onto the API calls. This allowed Felt, Chin, Hanna, Song, and Wagner to detect when apps were over privileged by looking for those with the permissions but not the associated API calls. The *PScout* tool [6] improved upon Stowaway. It did this by increasing the accuracy of the map between API calls and permissions. They built their map from the Android source code; whereas Stowaway used fuzzing.

API permissions are quite broad. The *internet* permission allows an app to send or receive anything on the internet. Several people have proposed a *finer grained permissions* *model*. For example: the internet permission could limit which addresses an app could talk to; similar to dependant typing.

The *RefineDroid, Dr. Android & Mr. Hide* tools [41] discover which permissions can be made finer, rewrite apps to use these permissions and then enforce them at runtime; they do this on a stock Android without needing rooting. Mr. Hide provides five new fine grained permissions:

IntentURL(d) allows apps only access to internet sites within the *d* domain.

ContactCol(c) lets apps access only certain fields from the contact information. For instance an email app might need to see contact information but wouldn't need telephone numbers.

LocationBlock forces apps to get location information from a special service that can mangle the location data arbitrarily; i.e. accurate to within a specified distance or shifted to a different location.

ReadPhoneState(p) forces the app to say which bit of information about the phone it requires and only grants it access to that.

WriteSettings(s) restricts which settings an app can write to.

The *AppFence* tool [40] doesn't modify apps. Users can write policies for what data an app can receive. If an app breaks this then it is stopped or fake data supplied instead. This requires changes to Android however. The *AppGuard* tool [7] rewrites apps to use a security monitor. This security monitor allows them to add extra checks when an app is sending or requesting data. They give examples for apps with the `INTERNET` permissions and show how they can restrict internet access. They show they can restrict access to given sites, force the use of the HTTPS protocol or block all network access; effectively removing the permission. AppGuard does not require rooting.

Sometimes a combination of permissions can be undesirable. Consider an app which has network access, starts on boot and which can access the internet. This app has the permission to act as a location tracker and could leak location information to an advertisement service, or a potential thief. *Kirin* [29] certifies apps at install time based on the requested permissions and potential dataflows between apps. Kirin lets users write security policies that prevent apps with certain permissions or intent handlers (discussed in Section 2.2). For example the location tracking app could be banned with the Kirin policy:

```
    restrict permission [ ACCESS_COARSE_LOCATION
                        , INTERNET
                        , RECEIVE_BOOT_COMPLETE
                        ]
and restrict permission [ ACCESS_FINE_LOCATION
                        , INTERNET
                        , RECEIVE_BOOT_COMPLETE
                        ]
```

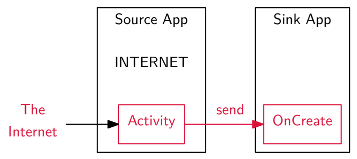


Figure 13: A flow between components a tool like SCanDroid might catch. The aim would be to detect that data from the internet is sent to an activity app which can then be sent to an app without the internet permission.

2.2. Intents and Collusion

Intents are the standard method for IPC on Android. This section explains the mechanism as well as the collusion attack to escalate a processes privileges.

Android uses a novel IPC mechanism called *Binder*. Apps use *intents* to share data and handle events. If an app wishes to handle an `SMS_RECEIVED` action it declares itself a *broadcast receiver* for the action; the app will be started when the event occurs. If an app wants to open a web page it can send an `ACTION_VIEW` intent. The user's browser will take open the URL. Apps can create their own intents. They can restrict usage of them to those signed with the same developer key.

Binder allows apps to collude to increase their privilege levels. Consider two apps communicating: one which can use the network and another which cannot. The unprivileged app asks the privileged app to send data on its behalf; the privileged app forwards the network responses back to it. The unprivileged app now has the network permission without declaring it to the user. If a privileged app does not secure its intents then they may break the protections offered by permissions. The *Kies* app by Samsung could be exploited like this to install other apps [49].

Tools have been made to find privilege escalation attacks. *Quire* [20] added origin tracing to intents. *SCanDroid* [34] statically analyzed apps to find flows across components. It describes constraints that should be satisfied to stop leaks.

TaintDroid [30] and *FlowDroid* [33] have been influential. Taint analysis is used to track data passed between apps. They detect when sensitive data is being leaked to an app. Others have shown that the approach is not perfect [53]: it can be defeated by malicious apps. FlowDroid takes a list of sources and sinks (found using the *SuSi* tool[51]) and tracks when the data from a source is sent to a sink.

```
// The Policy
policy ASSERTS
pgp:‘0x12345678’
WHERE PREDICATE =
regexp:‘(From: Alice) && (Organization: Microsoft)’;

// Queries
pgp:‘‘0x12345678’’
REQUESTS ‘‘From: Alice
Organization: Microsoft’’;

pgp:‘‘0x56781234’’
REQUESTS ‘‘From: Alice
Organization: Microsoft’’;
```

Figure 14: Example usage of the PolicyMaker authorization language.

3. Review of Policy Languages

3.1. Logics of Authorization

Logics of authorization allow policies to be written that restrict what an entity can do. They are a key topic in this proposal and various different logics are reviewed for comparison.

When an action is performed, such as reading a file or installing an app, conditions must be met for it to go ahead. The conditions form the *authorization policy* and we make a choice with respect to that policy when making a decision. When these policies describe what is needed to keep a secure system it is called the *security policy*. The policies can contain *trust* statements. Principals may be trusted to make statements about others and what is allowed.

PolicyMaker [14], grew out of the logics of authentication proposed by Wobber, Abadi, Burrows, and Lampson [44, 66]. PolicyMaker allows other principals (identified through asymmetric keys) to be trusted for actions or to declare further relationships. The language was minimal. It does not specify how the policies should be checked: they suggest regular expressions, or a special version of AWK. Any language could be used, however. The author suggested it might work well as a model for the public-key infrastructure. If we want a policy that only allows Alice (identified by key "0x12345678") to say she is Alice at Microsoft we write the policy in Figure 14. If we received a request from a different key to say they're Alice it would be denied; whereas a message from Alice's key would be authorized.

Checking whether a PolicyMaker policy is satisfied is NP-hard [15]. It is not tractable as checking PolicyMaker assertions can involve arbitrary programs written in Turing complete languages. Deciding whether a program will stop when given an arbitrary

```
Authorizer: ‘POLICY’
Licensees: ‘RSA:abc123’

KeyNote-Version: ‘2’
Local-Constants: Alice=‘RSA:123456’ // Alice’s key
Authorizer: ‘RSA:abc123’
Conditions: (app_domain == ‘RFC822-EMAIL’) &&
(name=‘Alice’) &&
(address=‘.*@microsoft.com’);
```

Figure 15: The PolicyMaker example rewritten in KeyNote.

input is analogous to the halting problem. So in general it is not known whether a PolicyMaker program which takes an arbitrary request and an unconstrained set of checking functions will terminate: it is undecidable. Blaze et al. give some restrictions that guarantee polynomial time checking: a function must be authentic (not fake another functions result), monotonic, and run in polynomial time for all inputs pertinent to a request. This reduced the expressiveness however.

KeyNote [16] is a revised version of PolicyMaker for public-key infrastructure. Like PolicyMaker it authorized actions based on keys and a series of conditions. It drops support for arbitrary program checkers; opting for its own specific language [17]. An example of KeyNote is given in Figure 15.

PolicyMaker and Keynote cannot express general statements where the subjects are not fully named. A store might have a policy that:

“Anyone who is a preferred customer and a student can get a discount.”

In PolicyMaker the key specified by the policy must be fixed: you cannot say any key with a property. For Keynote the local-constants have the same restrictions. Consequently these languages were not as expressive as hoped.

In comparison to KeyNote, *SPKI/SDSI* [28] is more complex. KeyNote claims this as an advantage over the SPKI/SDSI systems. Entities are described through name certificates. If Alice (with key K_{Alice}, had membership of the group `MSEmployees`; for a year; authorized by Microsoft (with key K_{MS}); she would present the name certificate:

$$(K_A, \texttt{MSEmployees}, K_{MS}, \text{1-year})$$

Microsoft could allow employees to send email for a year with the authorization certificate:

$$(K_{MS}, (K_{MS}\ \texttt{MSEmployees}), \bot, \texttt{send\_email}, \text{1-year})$$

Where $\bot$ indicates that delegation would not be allowed.

RT [45] builds on PolicyMaker. RT allows principals to be given roles; similar to a Role-Based Access Control (RBAC) system. Decisions are made based on which roles are held. RT can express the general statements that were impossible in PolicyMaker. For example consider the earlier example: students and preferred customers get discounts. To write this in RT for the Amazon store the policy would be written:

$$\begin{aligned} \mathit{Amazon.discount} &\leftarrow \mathit{Amazon.student} \bigwedge \mathit{Amazon.preferred} \\ \mathit{Amazon.student} &\leftarrow \mathit{Amazon.university.studentID} \\ \mathit{Amazon.university} &\leftarrow \mathit{NUS.accredited} \end{aligned}$$

RT statements are of the form "*Principal.role*"; where the first line of Amazons policy should be read:

> "Amazon says someone has the discount role if Amazon says they student and Amazon says they have the preferred role".

To claim the discount I would present the following assertions showing that Edinburgh is an accredited university and I am a student there as well as being an Amazon preferred customer.

$$\begin{aligned} \mathit{NUS.accredited} &\leftarrow \mathit{Edinburgh} \\ \mathit{Edinburgh.studentID} &\leftarrow \mathit{Joseph} \\ \mathit{Amazon.preferred} &\leftarrow \mathit{Joseph} \end{aligned}$$

If Amazon agreed with these assertions (i.e. they were cryptographically signed by the appropriate people) then it would grant discount.

Several versions of RT were described: the simplest being RT_0 [47] and with RT_1 and RT_2 adding support for parameterized-roles and logical-objects respectively. Extensions added support for constraints. A constraint is a limitation based on something outside of the policy language which may not be representable within it. This allowed RT_1^C[46] to express policies involving time (or other infinite sets).

The RT family is tractable as it can be translated into *Datalog* (specifically *Datalog with constraints*; also called *Datalog*^{C} *[46]*). Datalog is a query language similar to *Prolog.* Datalog does not support nested sub-queries or functions. It has a safety condition that all variables in the head must occur in the body. These constraints make Datalog a subset of first-order logic. Datalog queries can be answered in polynomial time with respect to the size of the knowledge base. Datalog is known to be tractable.

Cassandra [11] is influenced by the RT family of languages and Datalog^C. Cassandra was a trust management system used to model large systems. In his doctoral thesis, Becker showed how the NHS Spine could be formally modelled in the Cassandra language. The Spine is a complex and informally defined system: it describes the jobs and responsibilities of NHS employees.

```
  canActivate(mgr, AppointEmployee(emp))
  <- hasActivated(mgr, Manager()).
  canActivate(mgr, Employee(app))
  <- hasActivated(app, AppointEmployee(emp))
```

Figure 16: Role delegation in the *Cassandra* policy language. A manager is allowed to activate the employee role for an arbitrary entity by appointing them.

```
can(X, read, file) :-
  employee(X, company).
employee(X, company) :-
  hr says empolyee(X, company).
hr says employee(john, company).
```

Figure 17: Statements in *Binder* to say that in the current context only employees can read a file, and that an employee they must have a statement from HR to prove they are an employee.

In Cassandra principals activate and deactivate roles. Actions can only be completed if the principal holds the required roles. Delegation is allowed through an appointment mechanism. One principal can activate roles on other principals. Cassandra is tractable as it can be translated to Datalog^C.

The *Binder* language [25] was designed for authorization decisions [1]. It is implemented as an extension of Datalog. Properties are predicates. Predicates refer to entities. A *says* modality allows statements to be imported. If a predicate can be inferred from the knowledge base it is authorized. Binder does not add any predicates for handling state. The version of Datalog used does not allow for constraints. This limits Binder's usability.

3.2. SecPAL

SecPAL [12] is an authorization logic for decentralized systems. Syntactically SecPAL is similar to Binder. It has a richer syntax that allows for constraints; and it can make decisions based on state (such as the time). SecPAL was designed to be readable: it has a more verbose, English like, syntax than other authorization logics.

Like Binder it has an explicit *says* statement. Unlike Binder it requires that all statements are said by a principal explicitly. This ensures all SecPAL statements have an issuer; an explicit origin. SecPAL allows arbitrary predicates to be created. It adds two additional special modalities to the logic. The *can-say* statement allows for explicit delegation and has two varieties. The *can-say*$_\infty$ phrase allows for nested delegation, whereas the *can-say*$_0$ statement does not. This means you can distinguish between requiring a statement directly from someone and requiring a recommendation from someone. For instance if Alice wanted Bob to recommend a plumber directly she would write.

```
Alice says Bob can-say_0 person is-a-good-plumber.
```

In this scenario Alice would only be convinced someone was a good plumber if Bob told her. If Bob didn't know any plumbers but knew someone at work who had a used a few he might be tempted to recommend the work friend to Alice; Alice can get a recommendation from them and not need to come back to Bob to check whether he approves too.

```
Bob says Charles can-say_0 person is-a-good-plumber.
Charles says Diveena is-a-good-plumber.
```

Unfortunately Alice hasn't allowed delegation: so this wouldn't work. She will only be satisfied if she gets a recommendation from Bob directly. Bob may, of course, just reiterate what his work friend told him and satisfy Alice: the statement must just come from him directly.

Despite only having two delegation levels it is possible to express nested delegation to an arbitrary depth. To do this one imports statements which put a limit on how far the delegation can go. For instance if A wanted to allow B to delegate to someone who can delegate to someone but not any further than that; when importing statements A might opt to modify one to prevent further delegation.

```
A says B can-say_∞ x is-allowed.
B says C can-say_∞ x is-allowed.
C says D can-say_∞ x is-allowed. // Change this line
C says D can-say_0 x is-allowed. // to this line
```

Becker et al. sometimes write this using nested delegation statements; this syntax is non-standard as they disallow nested can-say statements as it complicates the evaluation making it potentially intractable (by breaking Datalog's safety condition).

```
// Non-standard SecPAL
A says B can-say_∞
  c can-say_∞
    d can-say_0
      x is-allowed.
```

SecPAL also adds a *can-act-as* phrase that allows *speaks for* relationships and entity aliasing. Suppose a user has a text messaging app SMSSender. The user also has a policy that they won't send a text message unless it is to someone in their address book. The user wants to try out a new messaging app, `SendSMS`, but still wants any restrictions from the old app to apply to it. Rather than duplicate all the rules for the new app (making their policy unwieldy) they can alias `SendSMS` to the `SMSSender` app. This ensures that the new `SendSMS` app will only be able to act if the old `SMSSender` could have. The SecPAL code for this is shown in Figure 18.

Extensions of SecPAL [10] add support for guarded universal quantification. They also remove the *can-act-as* statement. Other languages such as *DKAL* [38] built on and eventually split from SecPAL. DKAL was designed to express distributed knowledge between

```
User says SMSSender can-send-message-to(m)
  if m is-in-addressbook.
User says ContactsApp can-say₀
  person is-in-addressbook.

User says SendSMS can-act-as SMSSender.
```

Figure 18: Use of SecPAL's *can-act-as* statement to apply restrictions from one messaging app to another.

$$\frac{\begin{array}{c}(A \text{ says } \mathit{fact} \text{ if } \mathit{fact}_1, \ldots, \mathit{fact}_k, c) \in AC \\ AC, D \models A \text{ says } \mathit{fact}_i\theta \;\forall i \in \{1 \cdots k\}\end{array} \quad \models c\theta \quad \mathsf{vars}(\mathit{fact}\theta) = \emptyset)}{AC, D \models A \text{ says } \mathit{fact}\theta}\ \text{cond}$$

$$\frac{AC, \infty \models A \text{ says } B \text{ can say}_D \mathit{fact} \quad AC, D \models B \text{ says } \mathit{fact}}{AC, \infty \models A \text{ says } \mathit{fact}}\ \text{can say}$$

$$\frac{AC, D \models A \text{ says } B \text{ can act as } C \quad AC, D \models A \text{ says } C\ \mathit{verbphrase}}{AC, D \models A \text{ says } B\ \mathit{verbphrase}}\ \text{can act as}$$

Figure 19: The inference rules used to evaluate SecPAL. All SecPAL rules are evaluated in the context of a set of other assertions AC as well as an allowed level of delegation D which may be 0 or ∞.

principals by adding to the trust delegation mechanisms already in SecPAL. They also showed how any SecPAL statement could be translated into DKAL. The *SecPAL4P* language [13] was an instantiation of (the extended version of) SecPAL designed to specify how users' wished their personally identifiable information (PII) to be handled.

The inference rules for SecPAL are shown in Figure 19. Queries are evaluated against a set of known statements (the assertion context (AC)) and an initially infinite delegation level (D). If the rules show that the query is valid then SecPAL says the statement is okay else it is rejected.

SecPAL also imposes a safety condition on assertions. The safety conditions ensure that when the SecPAL program is translated into Datalog all constraints become ground (they do not contain variables). This ensures they're easy to solve.

$$
\begin{array}{rcll}
\mathrm{e} & ::= & \texttt{x} & \textit{(variables)} \\
 & | & \texttt{A} & \textit{(constants)} \\
\mathrm{pred} & ::= & \texttt{has} \mid \texttt{can} \mid \ldots & \textit{(predicates)} \\
\mathrm{D} & ::= & 0 & \textit{(no delegation)} \\
 & | & \infty & \textit{(delegation)} \\
\mathrm{vp} & ::= & \mathrm{pred}\ \mathrm{e}_1 \ldots \mathrm{e}_n & \textit{(verb phrase)} \\
 & | & \texttt{can-say}_D\ \mathrm{fact} & \\
 & | & \texttt{can-act-as}\ \mathrm{e} & \\
\mathrm{f} & ::= & \mathrm{e}\ \mathrm{vp} & \textit{(fact)} \\
\mathrm{claim} & ::= & \mathrm{f}\ \texttt{if}\ \mathrm{f}_1,\ldots,\mathrm{f}_n;\ \mathrm{c} & \\
\mathrm{assert} & ::= & \mathrm{e}\ \texttt{says}\ \mathrm{claim}. & \\
\mathrm{AC} & ::= & \mathrm{assert}_1 \ldots \mathrm{assert}_n & \textit{(assertion context)} \\
\mathrm{c} & ::= & \top & \textit{(no constraint)} \\
 & | & \mathrm{e}'_1 = \mathrm{e}'_2 & \textit{(constraints)} \\
 & | & \ldots & \\
\mathrm{e}' & ::= & \mathrm{e} \mid \mathrm{function}(\mathrm{e}_1,\ldots \mathrm{e}_n) &
\end{array}
$$

Figure 20: BNF specification of the SecPAL language.

First they define a fact to be *flat* if it does not contain a *can-say* statement.

```
Input  : a fact f
Output: a boolean indicating if f is flat
is-flat(f:Fact):
    match f :
        with $\star$ can-say $\star$ $\star$
            False
        otherwise
            True
```

A variable is *safe-in* a claim if a variable in the claimed fact f turns up in one of the conditional facts of the claim (fs) and not just the constraint.

```
Input  : an entity e and a claim c
Output: a boolean indicating if the variable e is safe in the claim c
safe-in(e:Constant, $\star$):
    True
safe-in(e:Variable, c:Claim):
    match c :
        with f:Fact if fs:[Fact]
            if e $\in$ Vars(f) :
                $\exists$ f $\in$ fs :
                    e $\in$ Vars(f)
            else
                True
```

Finally an assertion is *safe* if three conditions are met:

1. If the asserted fact is flat, all variables in that fact are also safe; otherwise the

delegated entity must be a safe variable or constant.

2. All variables in the constraint must turn up in the claimed fact or conditional facts.

3. All the conditional facts are flat.

```
Input  : an assertion a
Output: a boolean indicating if the assertion is safe
safe(a:Assertion):
    match a :
        with ⋆ says claim
            match claim :
                with f if fs, c
                    match f :
                        with x ⋆
                            condition-1 ∧ condition-2 ∧ condition-3

    where :
        condition-1 =
            if is-flat(f) :
                ∀ e ∈ Vars(f) :
                    safe-in(e, claim)
            else
                safe-in(x, claim)

        condition-2 = Vars(c) ⊆ Vars(vFs)
        condition-3 = ∀ f ∈ fs :
            is-flat(f)
```

3.3. Access Control Systems

Access control is an area where logics of authorization have been successfully applied. A review of several of the more commonly used schemes is given to contrast to the other authorization logics.

Access control systems control which users can have access to which files (or capabilities) on a system. These systems tend to fall into four categories:

Discretionary Access Control (DAC) where files have owners who control access to what they own. An example would be the standard UNIX and Linux permissions system.

Mandatory Access Control (MAC) where an administrator controls what users may or may not do with their files and whether they can disclose the contents to others. Examples would be SELinux, TOMOYO Linux SMACK or AppArmour

RBAC where access to files is granted on the to users holding roles; for instance only people working in personnel have access to employees records. Grsecurity is an example of an RBAC system and SELinux has some RBAC functionality too.

Rule-based where access is granted based on arbitrary rules: for instance someone in personnel may only access records during working hours.

Android uses two different access control systems: the DAC scheme traditionally used by Linux, and SELinux a MAC system developed by the NSA. The Tizen mobile operating system uses SMACK instead.

The DAC system used by Linux (and other systems descended from UNIX) is simple. Every file is owned by a user and a group. Permission can be granted (or revoked) by manipulating a bitfield associated with the file; this allows users to read, write or execute the file. These permissions can be granted to three sets of people: the owner, users in the group associated with the file, and finally any users on the system.

The DAC systems, such as Linux are problematic for secure systems as they allow users to control the files they own. This means that if a user is in control of what files get disclosed to other users. This is problematic for military systems as a malicious user might chose to disclose *top secret* information they own to a user without clearance. MAC improves on this by allowing an administrator to set the permissions and decide what can be disclosed by any user. The US *Department of Defence* mandated the use of MAC over DAC in their *Orange Book*[62] for all but the least secure systems.

SELinux is based on the Flask security architecture [56]. Entities are represented using classes such as *file*, *dir* (for directories) or *socket* (for network sockets). SELinux then defines operations that can be performed on the classes. The operations are more precise than Linux's DAC system (actually SELinux builds on the existing DAC system and only comes into effect if the DAC system would authorize a decision) and include operations such as creating symbolic links, appending and renaming. Users of SELinux have identities, and can be assigned roles that they are allowed to enter. Types (also called domains, a simplification from Flask) are what must be held to authorize a decision.

The SELinux policy file is made by concatenating together a series of policy files. These policy files consist primarily of type and allow definitions. For example consider Android SELinux policy files. By default apps run in the *appdomain* domain. To allow apps to read and write to the wallpaper file (which gives the image displayed on the home screen of the phone):

```
allow appdomain wallpaper_file:file { getattr read write };
```

The wallpaper file would be tagged with the `wallpaper_file` type in the filesystems extended attributes to associate the file with the tag.

If the policy author wanted to ban apps from setting system preferences, unless they were also running is some special system domains the rule would be:

```
neverallow { appdomain
    -system_app
    -radio
    -shell
    -bluetooth
    -unconfineddomain
}
property_type:property_service set;
```

SELinux can be complicated to configure. The policy language is implemented using the M4 preprocessor (which is somewhat arcane), and the policy file can be long: Android's basic policy rules are around three thousand lines long.

SMACK is a Linux security module (LSM), proposed by Casey Schaufler[55], that aims to simplify MAC configuration. It has been used in the MeeGo and Tizen mobile operating systems. Like SELinux it uses extended attributes to label files.

SMACK builds on the traditional DAC model allowing policies that describe who can read, write, execute and additionally append to files. If the labels of the process are a superset of the labels of the file the process wishes to access then the process is authorized to access the file.

The policy language is written in the form *subject object capabilities*. Several example use cases are given in the original proposal including an implementation of a read-down hierarchical security level system shown in Figure 21.

```
Classified  Unclassified  rx
Secret      Classified    rx
Secret      Unclassified  rx
TopSecret   Secret        rx
TopSecret   Classified    rx
TopSecret   Unclassified  rx
```

Figure 21: A hierarchical security policy for the SMACK access control system.

Role based schemes improve over MAC by shifting the capabilities from the users to the roles they perform; users can assume certain roles when they need to carry out work and shift to a different role after. This allows the policy to be more flexible as the privileges granted to a role are not defined for any specific user. Users can run with different limitations depending on the roles they currently hold; effectively sandboxing processes.

Grsecurity contains a RBAC system for Linux system[5]. It is used in some hardenened consumer electronics systems and some hardened Android devices. As well as enforcing access control decisions on files (including a pretend this file doesn't exist mode), it can also limit network connections, DNS resolution, capabilities a process can hold and stop certain kernel operations.

In the listing bellow an admin role is declared. It is a *s*pecial *A*dministrative role. All processes started by a user with this role[6] may *r*elax debugging restrictions, *v*iew and *k*ill all processes as well as *a*dministrate the system. For all files under the '/' path they may *r*ead and *w*rite files, *c*reate and *d*elete files or directories, *m*ark files as setuid

[5]Grsecurity is a rather large patch for the Linux kernel that hardens it preventing many attacks as well as an RBAC system.

[6]Technically all subjects (processes) started under the root file system which is almost the same thing.

or setgid, hardlink files, e*x*ecute or map into executable memory, and any new binaries *i*nherit being owned by the admin role.

```
role admin sA
  subject / rvka
    / rwcdmlxi
```

An unprivileged ssh daemon should be more restricted. It should run as a specific sshd *u*ser on the system, all files should be *h*idden, apart from the `/var/run/sshd` file. All capabilities have been disabled as has binding and connecting to network sockets.

```
role sshd u
  subject /
    /  h
    /var/run/sshd r
    -CAP_ALL
    bind disabled
    connect disabled
```

4. Review of Datalog

Datalog is used in the implementation of several authorization logics. A brief explanation of the language, how it is used in SecPAL, and its variants is given for some context as to how these languages can be implemented and the limitations the languages inherit from Datalog.

Datalog is a database language. It was created from a simplification of general logic programming. The language is based on first order logic; evaluation of Datalog is both sound and complete (under the closed world assumption (CWA)). Datalog is used as the basis for several of the authorization logics including SecPAL. We will review several evaluation strategies used for querying Datalog knowledge bases as most common method (bottom-up) is not particularly suitable for authorization logic applications.

Datalog programs are presented as series of Horn clauses in the same way as Prolog (see Section 4). There are additional restrictions, however: all variables in the head of a clause must be present in the body, and no parameter can be a nested predicate.

Datalog programs are split into two sets. The extensional database (EDB) has all ground (containing no free variables) facts. The intensional database (IDB) has rules for deriving more facts.

4.1. Evaluation Strategies

The *bottom-up* or *Gauss-Seidel* method is a simple evaluation strategy [21]. Given a Datalog program we try every constant with every rule from the IDB. When a rule is found to be true we add it to the set of facts. Repeat until a fixed point (or the required fact) is known. If a queried fact is still unknown when the search terminates then it is

```
person(alice).
person(bob).
person(claire).
person(david).
mother(alice, claire).
father(alice, david).
mother(bob, claire).
father(bob, david).
sibling(X,Y) :-
  person(X),
  person(Y),
  person(M),
  person(F),
  mother(X,M),
  mother(Y,M),
  father(X,F),
  father(Y,F).
```

Figure 22: A simple Datalog program and describing a family, and a relation describing what it means to be a sibling.

false; as Datalog assumes the CWA. The strategy is complete and will always terminate. Querying the database is fast once all facts have been inferred and large joins are quick.

This strategy ends up computing all known facts. It is less useful when only a subset are interesting. The *magic sets* [9] rewriting rule avoids this problem. Interesting constants are marked as *magic*. The knowledge base is a graph: nodes related to a magic one are also magic. Rules in the IDB are rewritten to check constants used in the inference are also be magic. This cuts down on irrelevant results: anything that isn't interesting will not be in the magic set.

The selective linear definite clause (SLD) resolution algorithm works top down. It starts with a goal and then constructs a proof tree. Transitions are applications of rules from the IDB. Nodes are either facts (the leaves) or further branches. If there is a subtree from the query node to true facts then it is true. Prolog uses this strategy. Its memory efficient as it searches the tree in a depth-first manner. Breadth-first and other tree traversal searches are also possible as are parallel strategies. The *top-down* strategy is less commonly used with Datalog programs. Saving previous search results (called tabling) is often used with this strategy to speed queries.

The SLD resolution may not terminate if there are a set of rules that set up an infinite loop (for instance the rule `a(X) :- a(X).`). Because Prolog has an infinite number of constants (integers for example) it is also possible to construct queries which return an infinite number of answers. Datalog does not suffer from this as it's programs must contain all known constants because of the CWA (and therefore there are a finite number

```
A says B can-say∞ X.
B says A can-say∞ X.
```

$$\dfrac{\dfrac{}{AC,\infty \models A \text{ says } B \text{ can-say}_\infty\, X} \qquad \dfrac{\dfrac{}{AC,\infty \models B \text{ says } A \text{ can-say}_\infty\, X} \quad \begin{matrix}\vdots\;\infty\\ AC,\infty \models A \text{ says } X\end{matrix}}{AC,\infty \models B \text{ says } X}}{AC,\infty \models A \text{ says } X}$$

Figure 23: A SecPAL assertion context, and partial proof tree that shows how an infinite loop would occur when evaluated with SLD resolution.

of them).

4.2. Datalog Evaluation in SecPAL

The bottom up strategy is commonly used with Datalog programs. Becker's paper describing SecPAL [12] points out that since their programs may change dramatically for every query, as statements may be added as the device runs in response to changing circumstances, recomputing all possible fact each time will not be efficient. The SLD resolution strategy is also not appropriate (despite Datalog's finite Herbrand universe) as SecPAL's *can-say* and *can-act-as* assertions could allow infinite recursion; as shown in Figure 23.

They present an algorithm for efficiently evaluating the Datalog. This is used with a Datalog translation of SecPAL programs. The algorithm uses the top-down strategy and tabling to speed inference. They also show the algorithm is sound, complete and always terminates.

To do this they construct a proof tree where each node is either a literal leaf p; or a tuple node consisting of a literal p, a set of subgoals qs, a constraint, the partial answer s, its children nodes nds and the rule used to construct the node rl. If a node has no subgoals and its constraint is met, then it is an answer node with answer s.

Two tables are also used: the *answer* table is mapping from literals to answer nodes. The *wait* table maps from literals to nodes which have not been fully answered. For a query p: $ans(p)$ and $wait(p)$ are the entries in each table pertaining to the query p.

To evaluate a query p for a given program with answer table ans the algorithm proceeds as in Figure 24.

4.3. Datalog Variants

Datalog does not support negation. It is not possible to write rules which depend on false facts. This is inconvenient as it is natural to write rules which rely upon a negative

```
# Evaluate a query against a program by checking
# first to see if we already know the answer,
# otherwise by resolving the query.
evaluate(p:Query, prog:Program):
    if ∃ p′ ∈ prog.answers: unifies(p, p′) :
        p′
    else
        resolve-clause(p, prog)

# Resolve a query by looking for a sub-query in
# the program that can be resolved (equal after
# renaming) and process that.
resolve-clause(p:Query, prog:Program):
    ans(p) ← {}
    for q ∈ prog.qs + prog.c :
        if ∃ nd = resolve(p,q +qs,c,[],rl) :
            process-node(nd, prog)

# When processing a node if it has no subgoals
# then start to process the answer, else start
# with the first sub-goal. If we know the
# solution to the subgoal after renaming then add
# it to the wait list, and start processing them.
# Otherwise add it to the wait list and work try
# to resolve the sub goal clause.
process-node(nd:Node, prog:Program):
    match nd :
        with (p, qs, c, ⋆, ⋆, ⋆)
            if qs = [] :
                process-answer(nd)
            else
                match qs :
                    with (q_0, ⋆)
                        if ∃ q′ ∈ prog.ans | can-rename(q_0, q′) :
                            wait(q′) $\overset{+}{\leftarrow}$ nd
                            for nd′ ∈ wait(q′) :
                                if ∃ nd″ = resolve(nd, nd′) :
                                    process-node(nd″)
                        else
                            wait(q_0) $\overset{+}{\leftarrow}$ nd
                            resolve-clause(q_0, prog)

# When processing an answer if we already
# didnt already know it add it to the list of
# answers. Then continue processing any waiting
# subgoals.
process-answer(nd:Node):
    match nd :
        with (p, [], c, ⋆, ⋆, ⋆)
            if nd ∉ ans(p) :
                ans(p) + = nd
            for nd′ ∈ wait(p) :
                if ∃ nd″ = resolve(nd′, nd) :
                    process-node(nd″)
```

Figure 24: Algorithm used to evaluate Datalog by SecPAL.

result: for example an app is safe to run if it is not malware. Whilst this extension to Datalog is not strictly required (the CWA can be used) it makes programs clearer.

A version of Datalog with negation called *Datalog*$^\neg$ [21] is made by allowing negation in clause bodies. Two sets of known facts are defined: those that are true and those that are false. When deciding if a fact is satisfied by a Datalog program if the fact is not negated then it must be inferable by the rules of the program; if the fact is negated then it must not be satisfiable.

In unmodified Datalog if the bottom-up strategy is used all possible facts are inferred. These facts form a single, minimal model of the Datalog program. In Datalog$^\neg$ the program safe(game) :- ¬ malware(game). has two minimal models that are inconsistent with each other: `safe(game)` and `malware(game)`. This can make analysis problematic as the CWA is broken. A further variant called *Stratified Datalog*$^\neg$ avoids this by further restricting what can be negated and defining an evaluation order [3].

Constraint Datalog (Datalog^C [46]) is based on constraint logic programming. Constraint logic programming allows relationships to be defined with general relationships (ordering by time for example)[7] rather than with just the pre-defined predicates. Being able to define relations in terms of general relations is convenient for authorization logics as it lets things be defined in terms of time or other general (and infinite) concepts.

An example of this might be this scenario. There are two guards who can open a gate: the day guard can open it from 6 am to 6 pm. The night guard can open it from 6 pm to 6 am. Another example is an access control policy that allows users to view all files within a directory.

Expressing these relations in Datalog is hard as the number of files within that directory or sub-directories could be infinite. The number of times in the watchmen's shifts is also infinite. Datalog would require each of these times and files to be instantiated. This is not ideal as it makes programs unwieldy. Policy languages, such as Cassandra [11], SecPAL [12] and RT_1^C [46] all use a form of Datalog^C as their evaluation engine to avoid this.

While some constraints applied to domains are tractable (such as trees, ordering and discrete domains) Li and Mitchell could not show all were. Policy languages that use constraint Datalog often apply additional restrictions on how constraints can be used. Variable independence conditions [23] have been suggested as a *middle-ground* as they can simplify the query evaluation while still keeping the extra expressiveness Datalog with constraints allows.

5. Work Done In First Year

In the first year of my study I have worked on developing an authorization logic that can express the user-oriented security policies for a smart phone [39]. Specifically the policies used when a user is installing apps. We have considered what kinds of policies

[7]Full logic programming languages like Prolog often support these relations. Datalog, however, does not as it would require all possible times to be named, and described in order to each other.

and trust relationships a user might wish to express and shown how they can be written in the language.

To do this we have been looking at a variety of authorization logics. These include BLF [65] and Binder [25]. SecPAL seems to be an ideal basis for the work as it is simple, extensible and readable. SecPAL's decentralized nature is ideal for describing a mobile-device and app-store ecosystem: there isn't a single authority making decisions about what can and cannot be installed onto a device.

We want to allow users to delegate decisions to experts. These might be third party certification or static analysis services; running on a remote server or on the device itself. Users should be able to use digital evidence [57] as a means of increasing trust in a tool. This might allow proof checking to be done with less strain on a mobile's battery.

We want to separate the checking of the user's security policy for the device (the *device policy*) from the policies any tool was checking for an app (the *application policy*). This meant that any analysis tool needn't use the same logic as the app checking tool. In the security policy static analysis tools are treated as oracles: they can utter statements about their inputs but we do not know (or care) how they came to these conclusions.

We extended SecPAL with two predicates:

`meets` The *meets* predicate says an entity believes an app meets an application policy.

The *application policy* is a property that Alice expects an app to have. This may be a rule that no personal information can be leaked over the network and she may expect this to be checked by another tool which could describe the policy in terms of the code. We do not need this second policy to be written in SecPAL, but we do need to name it so that the *device policy* enforced by SecPAL knows that it should or should not be enforced.

For example: if Alice believed the *AngryBirds* app met her policy to not leak information about her:

```
Alice says AngryBirds meets NoInfoLeaks.
```

We have not said anything about the *NoInfoLeaks* policy here, other than naming it and requiring *AngryBirds* to *meet* it. To check the policy Alice might describe a tool, that should be configured separately, that she trusts to make a judgement whether the policy is met.

Alice says CheckingTool can-say$_0$ app meets NoInfoLeaks.

`shows-meets` To express proof carrying code [50] and digital evidence we say that evidence *shows* a policy is met. We introduce the *shows-meets* predicate (whose notation we sugar somewhat). Consider again Alice who this time has managed to get digital evidence to show Angry-Birds won't leak her information.

```
Alice says Evidence shows AngryBirds meets NoInfoLeaks.
```

5.1. Alice Installs An App

To illustrate we describe a story where a user is trying to install an app.[8]

Suppose Alice has a smart phone. Alice has a security policy that says:

> "No app installed on my phone will send my location to an advertiser, and I won't install anything that Google says is malware."

Alice trusts Google to decide whether something is malware or not; or at least recommend an anti-virus vendor. She trusts the *NLLTool* to decide whether an app will leak her location. Alice is happy that if an app can come with a proof of it meeting a policy then she will believe it.

She translates her policy into SecPAL:

```
Alice says app is-installable
  if app meets NotMalware,
     app meets NoLocationLeaks.

Alice says Google can-say∞ app meets NotMalware.
Alice says NLLTool can-say0 app meets NoLocationLeaks.

anyone says app meets policy
  if evidence shows app meets policy.
```

Alice wishes to install Angry Birds. She downloads the app from a modified app store: apps come with statements about their security. Alice takes the statements and builds her assertion context. These statements include a recommendation from Google: McAfee can be trusted to decide whether an app is malware. There are also statements from McAfee and the NLLTool about the app itself. The assertion context is shown in Figure 25. Alice uses SecPAL to decide whether it says that `Alice says app is-installable.`

5.2. Implementation

We have implemented the SecPAL logic in Haskell and is around a 1000 lines of code, plus 500 lines of test cases.

In the original SecPAL paper [12] Becker, Fournet, and Gordon describe an efficient implementation using Datalog. We use a simple top-down approach. This was to quickly evaluate whether SecPAL is a good fit for the problem. It is not an efficient production ready inference engine. The program could not currently be used on a phone as most Android devices are poorly supported by Haskell compilers. It supports command history, dynamically loaded constraint-functions, comes with syntax highlighting plugins for Vim, and has handled simple assertion contexts with over a thousand statements. It is not ideal but can serve as a reference for a later efficient implementation if required.

[8]This example is built from work presented as a paper at the ESSoS Doctoral Symposium [39], and as a poster at the FMATS workshop.

```
Alice says app is-installable
  if app meets NotMalware,
  app meets NoLocationLeaks.
anyone says app meets policy
  if evidence shows app meets policy.
Alice says Google can-say∞
  app meets NotMalware.
Alice says NLLTool can-say0
  app meets NoLocationLeaks.
Google says McAfee can-say0
  app meets NotMalware.
McAfee says
  AngryBirds meets NotMalware.
NLLTool says ABProof shows
  AngryBirds meets NoLocationLeaks.
```

Figure 25: The full assertion context used to evaluate Alice's query.

An example of a proof generated by the tool is shown in Figure 26. The proof is presented as an inverted inference tree. Indented statements are the proofs for each condition of the unindented line above. Underlining indicates something is true as it either exists in the assertion context or is true in itself. Variable substitutions are shown in brackets to aid debugging.

5.3. Modelling Mobile Operating Systems

One example of a decision made by Android phones is whether to install or update an app. Android allows a user to install any app. There are two methods to install an app: through an app store, or *side-loading* (through a file manager or the Android debug bridge (ADB) utility). A request to install an app can be made explicitly by a user or programatically. An intent is broadcast to request the install the app. It is picked up by the package manager which confirms the request with the user (and are okay with the permissions requested). It checks the app has been signed correctly, and decides whether it need update or newly install the app. If this is okayed then the package manager passes the app to the *installd* service which does the actual installation on the file system. When it completes a message is broadcast by the manager to say the app has been installed (the `ACTION_PACKAGE_ADDED` intent) or updated (the `ACTION_PACKAGE_REPLACED` intent).

In the installation process there are several authorization decisions and delegations of trust. Firstly there is the trust given by the Android system to the package-manager (a service called `com.android.server.pm`) to manage the installation. All apps have a developer (identified by a public key within the app). This developer creates the signature on the app (which is checked by the package manager). We model this signature as a

```
AC, inf [app\AngryBirds] |= Alice says AngryBirds is-installable.
  AC, inf [app\AngryBirds] |= Alice says AngryBirds meets NotMalware.
    AC, inf [app\AngryBirds] |= Alice says Google can-say inf app meets NotMalware.
    ----------------------------------------------------------------------------
    AC, inf [app\AngryBirds] |= Google says AngryBirds meets NotMalware.
      AC, inf [app\AngryBirds] |= Google says McAfee can-say 0 app meets NotMalware.
      ---------------------------------------------------------------------------
      AC, 0 |= McAfee says AngryBirds meets NotMalware.
        AC, 0 |= True
        -------------
  AC, inf [app\AngryBirds] |= Alice says AngryBirds meets NoLocationLeaks.
    AC, inf [app\AngryBirds] |= Alice says NLLTool can-say 0 app meets NoLocationLeaks.
    ---------------------------------------------------------------------------------
    AC, 0 [anyone\NLLTool, ...] |= NLLTool says AngryBirds meets NoLocationLeaks.
      AC, 0 [evidence\ABProof] |= NLLTool says ABProof shows AngryBirds meets NoLocationLeaks.
        AC, 0 |= True
        -------------
      AC, 0 |= True
      -------------
  AC, inf |= True
  ---------------
```

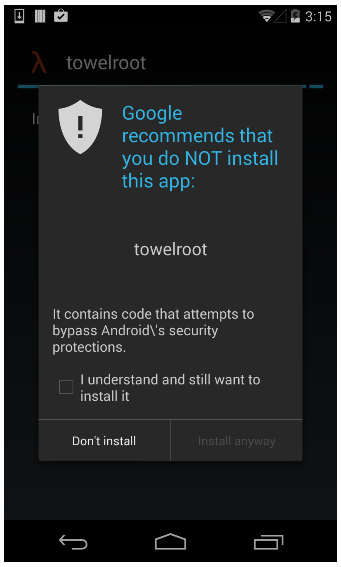
Figure 26: Proof output by the SecPAL tool when evaluating Alice's query.

statement by the developer that the app is owned; since SecPAL statements are signed assertions importing the statement into the assertion context would involve checking the signature matches the app itself. The package-manager asks the user whether the app should be installed: we model this as a constraint in SecPAL as this decision is made every time the app is installed and the decision made by the user is not stored as an assertion. Normally this prompt will only display the permissions the app requires, but for some app known to be dangerous and additional warning may be displayed as shown in Figure 27. The app is only installed if it isn't already installed, again we model this as a constraint. Finally the package manager only says the package was installed if the installation process carried out by `installd` succeded.

```
Android says Com.Android.Server.PM can-say0 app PACKAGE_ADDED.
Android says Com.Android.Server.PM can-say0 app PACKAGE_REPLACED.

Com.Android.Server.PM says dev can-say0 app is-owned;
  developer(app) = dev.

Com.Android.Server.PM says app PACKAGE_ADDED
  if app is-owned;
     Installed(app) = False,
     UserInstallPrompt(app) = ``Okay'',
```


3:15

towelroot

Google recommends that you do NOT install this app:

towelroot

It contains code that attempts to bypass Android\'s security protections.

I understand and still want to install it

Figure 27: The additional warning displayed when a user attempts to install an app known to be dangerous: in this case *George Hotz's Towelroot* app that enables access to the root user account through a Linux kernel exploit. `http://towelroot.com`

```
    installd(app) = Success.
```

The update process has additional restrictions: the old version of the app must have the same developer (more precisely it must have been signed by the same key), and the version number must be higher. If there is no change in permissions (or they're reduced) then this is sufficient for the app to be installed, otherwise the user is prompted for a confirmation that the update is okay again.

```
Com.Android.Server.PM says app PACKAGE_REPLACED
  if app is-owned,
    app updates(old-app);
    Installed(app) = True,
    developer(app) = developer(old-app),
    version(app) > version(old-app),
    permissions(app) ⊆ permissions(old-app),
    installd(app) = Success.

Com.Android.Server.PM says app PACKAGE_REPLACED
  if app is-owned,
    app updates(old-app);
    Installed(app) = True,
    developer(app) = developer(old-app),
    version(app) > version(old-app),
    permissions(app) ⊃ permissions(old-app),
    UserInstallPrompt(app) = ‘‘Okay’’,
    installd(app) = Success.
```

6. Thesis Proposal

In my first year I have been learning the SecPAL language, evaluating what scenarios in Android it can express, and trying to find the right logic language to model the issues surrounding smart phones. The next two years will be spent exploring what happens when these policies interact with users.

In the remaining time I will work on the following areas:

- Describing how authorization decisions are made in Android using an authorization logic.
- Building an app store that allows for the filtering and modification of apps on the basis of policies.
- Evaluating how different policies interact and the effects of filtering.
- Creating richer policies for users and to describe difficult Android scenarios.

6.1. Policy Language for Mobile Devices

The first stage of the project will be to complete the work done in the first year. I will show that a logic of authorization can model the security decisions made inside Android; and that it is capable of describing complex security policies.

The logic of authorization must be able to describe how the current mobile ecosystems (such as Android and iOS) work. There are already several situations where Android makes a decision if an app is allowed to be installed or updated. The reasoning behind these decisions is embedded in the code itself. The authorization logic will describe these policies: this will clarify and formalize the policies present in the code.

The description of the Android installation procedure in Section 5.3 suggests that the SecPAL language could describe the authorization decisions in Android. We have been able to describe the installation and update scenario with relative ease. Installation is not the only decision we have to make on Android. Other areas include:

- Comparison of different app stores app submission policies.
- Android permissions system.
- Android’s SELinux policies and processes.
- The remote procedure call (RPC) calls and Intents between apps.
- Data sharing between apps.

- Remote data access and debugging.

This is interesting because by modeling these decisions we are able to compare the different mobile OSs precisely. It also may allow us to find problems in the implicit policies in the code by specifying them formally.

To describe these scenarios we need a policy language that has the following features:

- Model the decisions at separate locations. This allows us to check the policies locally and make a decision based on that location's information rather than deferring to an all-seeing policy enforcer. This suits the problem better because individual devices and stores have to make decisions on their own and may not have access to all known information.

- Explicit speakers that can be checked. In practice this means some form of signature scheme on assertions. This is important as it will allow us to be sure that our statements come from the correct place. We plan to use the language to make decisions that have security implications (such as installing software) based on information we may get from the entities (like an app store) that we access through a network. Not being able to trace the origin of statements seems dangerous.

- Calling external functions (such as the `UserInstallPrompt` constraint used when describing installation) that can obtain extra information about entities that may require special actions to fetch: for example an explicit "okay" from the user, or some metadata about an app like its version number. If we don't allow the ability to fetch this information on-the-fly then it will have to be requested before checking the policy. In the case of the `installd` function, where we check if the actual installation succeeded, this is especially important lest we install the app before we have made the decision whether to install the app or not.

  Alternatively *obligation policies* [35, 36] could be used. Obligation policies are used to describe a set of tasks an entity is obligated to perform in the future to make a statement. For instance, in the case of package installation we could make the actual installation of the package (the `installd(app) = Success`) an obligation to perform if the package manager wanted to state that the package is installed. These kinds of policies do not seem to fit the app installation scenario well, as we would want the statement that the app is installed to be emitted only after the app is actually installed. That said obligation policies have been used in the KAoS [63], Ponder [24, 61], and Rei [42] distributed-systems policy languages and may be a useful strategy here too.

- Support for constraints. Android apps have version numbers (for example) that will need to be compared with other version numbers. If we cannot compare numbers and other infinite-domains simply then policies will become tricky to express. Constraints allow us to make these kinds of comparisons; this will allow us to write the policies we want.

	Policymaker	SPKI/SDSI	Cassandra	RT	Binder	SecPAL
Local checking	✓	✗	✓	✓	✗	✓
Explicit speakers	✗	✗	✗	✓	✓	✓
External functions	✓	✗	✓	✗	✗	✓
Constraints	✗	✗	✓	✓	✗	✓
Delegation	✗	✓	✓	✓	✓	✓

Figure 28: Table showing each of the features we described as being desirable and the authorization language described in Section 3. The table compares which languages support which features.

- Delegation. We want to make a decision based on information from an external source. This suggest that a *can-say* mechanism is required that will allow us to express how trust is distributed. Since the relationship between app stores and devices is one where a user delegates trust to the store and the device uses the external store to obtain the app this would hint that being able to express delegation will allow us to write more accurate policies.

Ideally we would also want checking policies to be efficient and to not drain the batteries. We want to use the language to be able to make decisions on a phone, which do not typically have large memory and where battery life is at a premium. If we drain the battery whenever we make a decision then users will be excessively inconvenienced. Whilst we are modelling the decisions on Android this is less important (the logic can be run on a more powerful system) but it may be a concern later.

A comparison of the features is shown in Figure 28. Based on these requirements SecPAL seems to be a good fit; it has certainly been able to describe the scenarios we have encountered so far (though some hypothetical policies that SecPAL cannot express well are described later on page 48). Alternatively Cassandra, which like SecPAL was developed by Becker, or one of the RT languages might work. Cassandra's policies, however, involve entities activating roles which feels less natural for describing policies that describe user preferences and software processes. The RT family of languages lack external functions, though these could be added.

I plan to describe fully the security of Android authorization decisions. The report will describe the authorization logic, why it was chosen over other policy languages. How common decisions made by mobile operating systems may be described in it; how policies users want (such as the late-adopting and hipster policies) may be expressed in the language; what problems we may encounter and their potential solutions. This technical report will be the first milestone for my PhD and will be used to provide the

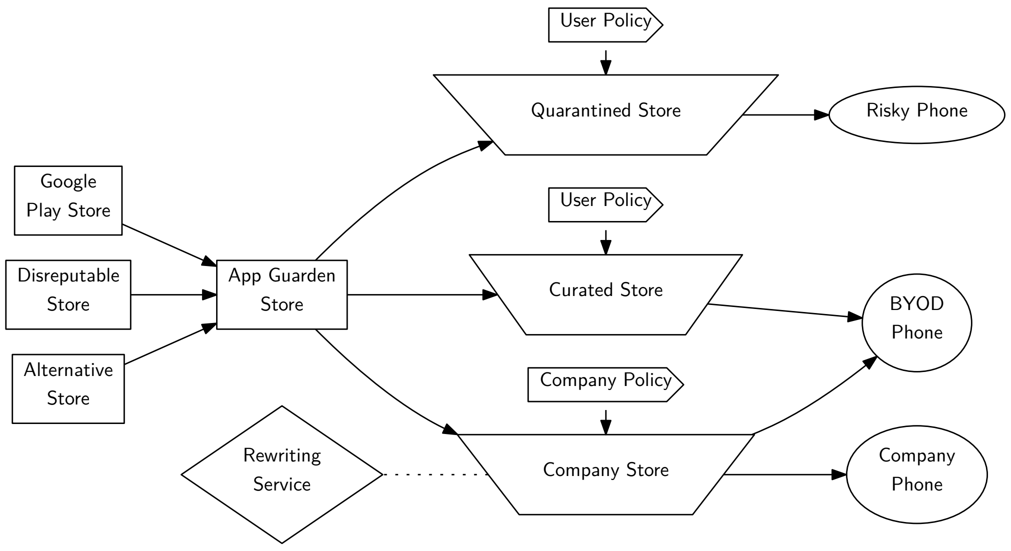


Figure 29: Model of App Guarden store architecture.

start to my thesis.

6.2. Exploring Policies and App Guarden

The next area to look at will be the policies users want when downloading apps. To explore these policies we will create an app store where users can express their policies. This store will be the *App Guarden* and be used as a play ground for security policies and Android analysis tools.

The store will use security policies to filter apps. Creating an app store allows interaction with my research. Encouraging users to use an app store with security policies increases the impact of the research. It provides a practical example to illustrate how it can be applied to a real world problem. It also will offer a platform to test real policies against and show how different analysis tools can make different guarantees.

To build the store we will base it on the proxy store model shown in Figure 6. The final store architecture will resemble Figure 29. We will download apps plus metadata from the Play Store into our own *App Guarden Store*. From this master store we will create start to filter apps on the basis of policies. The results from this filtering will become the *curated stores*.

The store will act as a framework to compare different stores policies with. By checking policies in the app store it avoids needing for users to root their phone[9]. It also allows

[9]This can reduce their devices security as it allows malware potential access to the root account. It is a known infection strategy by malware [58].

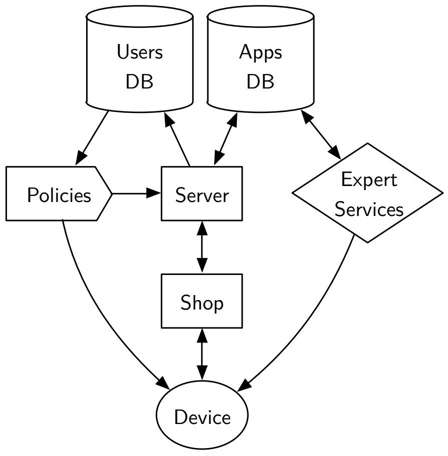


Figure 30: Components which speak to each other in an implementation of the App Guarden curated stores.

for a wide range of users to interact with the project.

Creating a secure app store is a non-trivial engineering challenge; especially when combined with the danger provided by user supplied policy files[10]. The engineering difficulty in developing such a store can be mitigated by sensible software development practices. Android allows apps to be installed by the system installer; after confirmation by the user. To implement the *shop* one could embed a website in an app that downloads and opens apps from an external *server*. The server can provide apps to the device from an *apps database* and allow *users* to register themselves and their *policies*. Finally apps can be sent to *expert checking services* who can send their results to the app database and the *device* wanting to check their statements. This is shown in Figure 30.

These curated stores will give us several areas for investigation:

Policies and users: Policies should affect which apps are available to the user. If an app cannot be reconciled with a user's policies then the user should not be able to install the app. Simply preventing the user from installing (or even seeing that an app is available) is all very well but how will users react to being told they cannot install their app? Will they understand the reasoning why an app has been denied?

Users routinely circumvent security measures [18]. We don't want users to remove protections, and we do not want users to write all permissive policies. Finding a balance between writing restrictive security-enhancing policies, whilst still giving users a sense of control and freedom, is necessary if authorization logics are to be a practical tool.

[10]Interpreting user supplied data can be dangerous as if the interpreter can be exploited through the data passed, this can compromise the integrity of the entire store. Similarly if the policy language can form a *weird machine* [19] then it is possible to tie up the interpreter indefinitely or perform arbitrary calculations.

```
Phone says app is-acceptable
  if app meets NotOverPrivileged.

Phone says Stowaway can-say0
  app meets NotOverPrivileged.
```

Figure 31: Policy to filter over-privileged apps.

To explore this we will develop a hypothetical policy for a company and create a curated store where the policy is enforced. The effects of different policies will be measured by the number and kind of apps in the *company store*. This will let us measure the restrictiveness of policies on real apps, it will allow us to compare different checking tools, we will be able to test theories about different clusters of apps. For instance if we had the theory that most flashlight apps are over-privileged we can check it by writing a policy that rejects apps if Stowaway [31] says they are (Figure 31): if half or more apps categorized as flashlights are blocked then we have evidence the theory is correct.

To test how users will interact with the store a user study will look at how well users comprehend their security policies and their apps. One question we could answer is *do users understand why an app is unsafe?* To test this we could give users a policy, a decision made by a policy enforcer, and a justification: the user would then be asked if they think the enforcer made the right decision. If the users agree with the enforcer and justification (when they are correct) would suggest that users understand the policy enforcement. Checking if users then still install the app would test whether users sufficiently understood the risks (or just didn't care).

We will also start looking at what happens when policies and apps change. A security policy is not a constant document. Policies require updating when circumstances and risks change as well as when a policy needs relaxing or tightening for a specific app. Apps are updated by their developers. Sometimes these apps need more permissions for new features. As was discussed in Section 1.2; it is not alway clear what to do in these circumstances. Several solutions are available:

- Remove the apps.
- Run the apps in a restrictive sandbox.
- Prompt the user for exceptions to the policies.

None of these solutions are ideal. By testing these solutions against devices using apps we control we can find exactly what the trade-offs are. Which solutions do users prefer and what are their reactions? If the policy is going to be broken what aspects get broken most often? If the breakage causes information to leak, how much information? Other solutions may present themselves: and user's (or at least administrators) should be able to choose between them with knowledge of what that entails.

Policies and apps: Can different categories of apps be filtered by policies? Are there policies that apply to one category of apps but not others? We might expect that social apps have access to your contacts but not flashlight apps.

To test these kinds of policies we need to be able to categorize apps. Google's PlayStore organises apps into sections such as *games* or useful apps. If a policy we write to describe games identifies with a high probability apps in the games category then we should gain confidence that the policy is good. Alternately if an app that purports to being in one category doesn't meet that categories policy then maybe the policy is wrong, maybe the category is too general, or perhaps the app is malicious!

The App Guarden store will allow us to check policies such as these by providing a set of apps and a filtering mechanism. Similar to the company store we might create a games store that only sells apps we categorize into games on the basis of policy. By comparing the apps that in these policy-specific stores against other app stores we can gain insight into how different app stores work. For example if the PlayStore categorises games apps on the basis of one policy, but apps in Amazon's store games category are best described by a different policy then we can contrast these two policies to learn about how the app stores themselves differ.

Compound policies: In Section 1.2 we introduced the idea that a policy on a device may be formed from several policies. For instance an employee may use their own personal phone for work too. This is the BYOD scheme described earlier in Section 1.

Compound policies have some interesting problems surrounding them. Suggesting solutions to these problems, and evaluating them is novel; existing work on security policies looks at when there is just one policy.

To explore compound policies we will suppose a scheme where a phone is used at home and at work. The device must follow two policies: the user's personal policy describes how the user expects the device to behave; a corporate one describes how the device should behave at work. Under this scenario several questions can be asked:

- *How are policies composed?* Is there only one strategy for composing them? A simple strategy might be to always take the more restrictive policy if there is a conflict but otherwise ensure it meets both.

  Alternatively you may accept apps which meet either policy but introduce additional restrictions on them The *Qubes OS* [52] (a hypervisor operating system for X86 computers) separated apps needing to run with different security policies into separate virtual machines. Approaches relying on separate virtual systems for different composed security policies are becoming more viable on mobile platforms; this may be preferable to policy composition.

What happens when the policies disagree? A user may have a policy that they don't want their location exfiltrated; their employer, however, may require that they can track their employees location during work hours. In this scenario we can't take the more conservative policy and not leak any information as this disagrees with the corporate policy. We can't sandbox the different apps as this will still leak the location to the employer and break the user's policy. Scenarios like this suggest policy composition is not a simple problem.

- *Reusing Proofs* When the policies are composed can you show that an app meets both policies together if you have proofs it meets both policies separately? Intuitively the answer seems to be yes: if you know policy A is met and policy B is met then policies $A \wedge B$ should be met too. Being able to show this, and generate certificates for their proofs might make checking policies more efficient as the less work would have to be done by the inference engine (and potentially the static analysis tools).

  What happens when policies agree? Suppose a new, stricter, policy is required. The developer takes a pre-existing policy and adds extra rules to it. Later this new policy is composed with the old. Apps that met the new policy will already meet the old one so do we need to do any extra checking at all?

- *What happens to existing apps?* Polices on devices should be expected to change over time. A new policy could be made by composing a new set of rules (from a company) with an existing set (from the user): but what should happen to the users existing software? In Section 1.2 several possibilities were discussed; none of which seemed satisfying. The App Guarden store will allow us to explore and find the consequences of each of them.

Policies under attack: Security policies should prevent apps behaving undesirably. Security doesn't remain constant, however, and as defences become more popular malware authors search for new ways to circumvent them. This aspect of the store will look at what happens here.

There are several ways that a security policy could be circumvented:

- *Misplaced trust.* Checking policies requires trust in other entities. Suppose we trust a tool to decide whether an app has a security property. What happens when the tool gets the decision wrong? There are often limits to what analysis tools can check [48]. Alternatively the owners of a tool may be coerced or bribed into giving an incorrect answer.

  This kind of failure leads to decisions being authorized incorrectly. How this affects the security of the device and knowing what must be done to recover from it is important for policies to be resiliant. A variation of this might be service which stops decisions being authorized. For example, suppose a video streaming service bribed an antivirus service to say all its competitors were malicious. Knowing how to recover from this (especially after a user may have installed the video streamer out of frustration) is not obvious[11].

- *Policy is incomplete.* A user might have a policy that their address book should not be leaked; and they might enforce the policy by using taint analysis to check all data being sent out over the internet. If a malicious app is distributing the contacts by Bluetooth or text-message then the analysis may not catch it. This can be particularly annoying for users as they can have a simple policy but the checks needed to enforce it can be long and technical.

  By developing *malware* which aims to slip past security policies we will be able to find better policy idioms. This will lead to a more reliable policies.

- *Exploitation.* No software scheme will be without bugs, and sometimes these bugs can be exploited. If someone can crash or control the policy checker it cannot be trusted to decide policies. This is hard to prevent, but some thought should be given to it.

Testing for these issues can be difficult as it is hard to use real apps without checking they will or will note meet a policy by hand; and synthetic malware and PUS may be trivially detectable[12]. It is important to consider, and if possible evaluate, these failures as they are an obvious way to attack a system. Any claim that authorization logics will solve all the problems in mobile devices are overinflated [43].

6.3. Richer Policies

So far we have described policies that describe existing mechanisms, and policies that use analysis tools and metadata to make judgements about apps. This final section of the project will involve creating more complex policies that describe both Android specific problems and also policies that a user might want to enforce that are richer[13] than previously looked at. These policies will then be tested on devices using the app store framework.

One topic will be to show how policies can be written to take into account of the *app collusion problem.* The app collusion problem is shown in Figure 32. A user has a policy that no app that can access the contacts should have access to the internet. A device has two apps, the first can read the contacts, and the second can access the internet. Individually they both meet the policy, however if the first app was to send the contact information to the second app (through an RPC mechanism like Binder's Intents on

[11]Though perhaps a similar solution to Microsoft's Internet Explorer being compelled to advertise alternatives might work well.

[12]*Schneier's Law* [54] states that:

> "anyone can invent a security system that he himself cannot break."

[13]That is to say policies which are more complex than showing an app gets a specific result when analysed by a series of tools, or with certain app specific metadata.

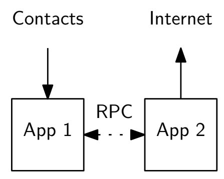


Figure 32: Two apps who could collude to leak contacts over the internet

Android) then the second app could take that data and leak it breaking the policy. A taint analysis tool, like *TaintDroid* [30], can detect when this attack occurs.

Whilst there are tools to detect and attempt to prevent these kinds of attacks there has not been an attempt to model the decisions to collude and with whom in a logic of authorization. Collusion is not, in itself, a sign of malicious intent. For example consider a policy that says:

> "an app shouldn't send who I follow on social network to another app."

Suppose a user wishes to save something posted by a friend to an online bookmarking service: rather than saving just the link they save a link to the post itself; as is common on social networks like *Twitter*. The app has broken the policy now as the bookmarking service is aware that the user follows the friend. Whilst this is an information leak that breaks the policy; it is one the user intended. It might be prudent to remind the user that this will tell the bookmarking service information about them; stopping the action will likely annoy the user.

Another area to look at to do with policy language will be to ensure the language is flexible enough to handle different scenarios when handling updates to applications. This will include looking at the whether permissions increase over time or if developers actively prune the lists; as well as developing policies that can describe what to do when a well-used app no longer meets the security policy.

If SecPAL is to be used as the authorization language then there are also some policies which might be problematic to express in it. Consider the *late-adopting* user who will only install an app if two or more of their friends say it is good; or the *hipster* user who will only install an app if no one else is using it. In SecPAL the one approach would be to say this is similar to a *threshold signature*; a scheme where a signature is only valid if it has been signed by several signatories. In Becker, Fournet, and Gordon's paper [12] they describe a scheme for threshold signatures in SecPAL that we could apply here. The late-adopting user would say:

```
User says x can-say₀ app is-recommended-by(x)
  if x is-friend.

User says app is-good
```

```
  if app is-recommended-by(x),
     app is-recommended-by(y);
     x ≠ y.
```

This seems to work but there are some problems: firstly we cannot express the *hipster* case like this as we don't want anyone to say the app is recommended and SecPAL cannot express this. Secondly the phrasing is clunky as this relationship is very similar to SecPAL's *can-say* relationship (which cannot be used here as conditional facts don't come with an origin). It also isn't obvious how a friend can delegate the recommendation to one of their friends as they might do with the *can-say* relationship. This seems natural: a conversation might be *"Do you recommend this app? Oh I think my friend uses it why don't you ask them?"*.

If we were to alter SecPAL's syntax and allow conditions with an explicit origin we could write:

```
User says x can-assert∞ app is-good
  if x is-friend.

User says app is-good
  if x says app is-good,
     y says app is-good;
     x ≠ y.
```

This seems more natural but requires changing the language. We need to add a new *can-assert* rule that, like can-say, allows delegation but doesn't make statements true unless used in a conditional statement. These changes may invalidate the safety and termination proofs from the original paper [12] which is undesirable. This still doesn't account for the hipster case.

A third alternative would be to borrow the *find-all* construct from Prolog. Find-all allows the programmer to find all the statements that satisfy a given predicate. For example to find all of Alice's cousins one could write in Prolog:

```
  findall(X, cousin(Alice, X), cousins).
```

A similar construct could be brought into SecPAL:

```
User says x can-assert∞ app is-good
  if x is-friend.

User says app is-good
  if findall(_, app is-good, recommenders);
     sizeof(recommenders) ≥ 2.

Hipster says app is-good
  if findall(_, app is-good, recommenders);
    sizeof(recommenders) = 0.
```

This accounts for the hipster case and helps keep policies short (one could imagine a user saying that it wouldn't install an app unless a million people recommended it which would lead to unwieldy policies in the earlier versions). This changes the language dramatically: `findall` isn't a Datalog instruction; this may not even be translatable.

Another problem occurs when we are using a checking tool. Suppose we have a tool which can infer complex properties about apps; for particularly complex and large apps however the checking becomes infeasible[14]. You could imagine a security policy for a phone where the user uses the tool on some small apps but for large apps the user is allowed to make the judgement:

```
Phone says FlowDroid can-say₀ app meets FlowPolicy
  if sizeof(app) < 500MB.

Phone says User can-say∞ app meets FlowPolicy
  if sizeof(app) ≥ 500MB.
```

The `500MB` is a rather arbitrary constant here. If we wanted to change the policy it would need to be changed in multiple places. A better solution might be to allow *FlowDroid* to answer that it doesn't know about the app (perhaps it ran out of resources) and then make a decision based on that.

```
Phone says FlowDroid can-say₀ app meets FlowPolicy.
Phone says FlowDroid can-say₀ app unsure-of FlowPolicy.

Phone says User can-say∞ app meets FlowPolicy
  if app unsure-of FlowPolicy.
```

Suppose the user decided to make an exception and allow the app. Later, after FlowDroid is given access to more memory perhaps, FlowDroid analyses the app and finds proof that the app does not meet the policy. FlowDroid revokes its statement that it is unsure of the app meeting the policy, and the phone no longer believes the app meets the policy.

In this case it works, but suppose instead the user decided to trust two different tools to decide whether the policy is met. Both tools initially say they're unsure if an app meets the policy; the user provides the exception. Later one of them revokes its statement. This time the phone still believes the app meets the policy as it still has the override and the statement about one of the tools being unsure.

This suggests that using the CWA may not be a helpful assumption. An alternative would be to allow negation in SecPAL (and use Datalog$^\neg$). Adding an explicit *breaks* predicate instead of *unsure-of* we write:

```
Phone says User can-say∞ app meets FlowPolicy
  if ¬ app meets FlowPolicy,
     ¬ app breaks FlowPolicy.
```

[14]For instance in a meeting with developers of *FlowDroid* they claimed their tool could analyze the Facebook app, but took 75GB of RAM to do so. Whilst this was feasible on a compute cluster, this could not be done on a mobile phone.

This changes the language however; which again may lead to invalid safety proofs.

It also raises questions about when two trusted tools disagree. A cautious approach would be to always follow the *breaks* statement: if there is doubt then trust no-one and reject everything. With static analysis tools this might not be the right approach: false warnings are common amongst some tools (as anyone who has used the *lint* tool can attest). Suppose a malicious developer bribes a certification tool to be more strict with their competitor (or even just to devote less resources to checking it): they would create a denial-of-service for the app.

Asking the user to pick between the tools isn't a good solution either. Suppose *Norton* and *McAffee* disagree about whether an app is malware or not. Unless the policy author is skilled at malware-analysis they may not be able to decide who to trust here.

The solution to these problems, amongst others, is not immediately obvious. That basic Datalog, and therefore many policy languages, may not be able to describe possible solutions to the problems suggests the work is non-trivial.

6.4. Possible Extensions

This project is expected to take three-and-a-half years. If the project is completed before that time then there are several extensions that could provide interesting extra work.

Android uses SELinux to enforce restrictions on apps. Currently Android app processes are run in several domains: the *appdomain* (a domain for apps started from the Zygote process), *system_app* for apps run as the system user, *untrusted_app* which is used for apps by default and any system apps not signed by the platform key, *platform_app* for apps signed by the platform key, and *isolated_app* for apps which declare they should run without any permissions and limited communications [37]. Access to specific functionality is controlled by the Linux DAC.

These SELinux domains are declared statically in the Android policy files: however this needn't be so. SELinux allows the policy to be changed at runtime. An app, that perhaps would not meet the user's device policy, could come with a set of SELinux rules that add additional restrictions to the app to ensure it met the device policy. An alternative would be to generate rules for an intrusion detection system (IDS) that could detect when the authorization logic has failed.

This would increase the trust in the system as it adds additional run-time restrictions that enforce the policy. This is on top of the checks that the policy is met. Generating IDS test would aid debugging as it would identify cases where the authorization logic has identified a false positive (i.e. a policy breaking app that was allowed onto the device) and could become digital evidence that the app did not meet the policy; implying trust has been misplaced in the checks.

Another area would be to develop a protocol for distributing knowledge. Whilst SecPAL describes informally a method for transmitting *says* statements (they're signed by the speaker's asymmetric key). When an entity wishes to use this statement it gets the public key of the speaker and checks the signature matches the message (as shown in Figure 33).

```
says(s:speaker, m:message):
    $\sigma \leftarrow$ sign(m $\|$ s.id, s.k)          # s.k is the speaker's secret-key
    return(m $\|$ s.id $\|$ $\sigma$)

hears(m:message, K:Key-store):
    match m :
        with (m' $\|$ id $\|$ $\sigma$)
            k $\leftarrow$ K [ id ]                       # k is a public-key

            if check(m' $\|$ s.id, $\sigma$, k) = True :
                return(s.id says m')
```

Figure 33: Method for speaking and listening in SecPAL.

This describes the mechanism behind the says mechanic; but does not say how the information is actually transmitted. It does not describe how the listener gets the message and why the speaker said it in the first place. The mechanism for distributing redactions is not described.

One way to do this might be with a framework similar to a public-keys infrastructure. Statements can be stored at a central repository (with redactions), with a key-store.

This might work for tools and in an app-store context; but when queries are made by private individuals they might be discouraged from making publicly available statements about apps they have on their phone. For example someone with a medical condition might not wish to say they have an app to monitor it. When a user's policy is to only install apps that their friends already used (as discussed in Section 6.1) how do they ask their friends for this information?

Can we extend SecPAL to express a policy that decides when an entity will distribute information? One possible way to do this would be to add rules to SecPAL that describe what information an entity can-ask; this would be an extension of SecPAL's syntax similar to the *can-say* rule. Taking the earlier example a user will allow friends to ask whether they think an app is good or not; but only if the app isn't a dating app:

```
User says x can-ask-if app is-good
  if x is-a-friend;
     category(app) ≠ 'dating'.
```

Devices, or other principals such as stores, would have an API to allow queries to them. When a statement is requested the entity would check if the *can-ask-if* phrase is satisfied before distributing the statement asked for or a *no-response* response. If SecPAL had been extended to allow negative statements then one could also be returned; allowing a querier to distinguish between there being no response and the principal not being willing to say the queried statements. This may have privacy repercussions however as a querier may be able to learn the queried party's information distribution policy by looking at which queries it refuses to answer.

```
I say Alice can-say₀ app is-not-dangerous
  (with confidence 90%).
I say Bob can-say₀ app is-not-dangerous
  (with confidence 60%).

I say app is-installable
  if app is-not-dangerous;
     confidence > 75%.
```

Figure 34: Example of how levels of trust might be expressed in SecPAL.

Another area to look at would be to quantify the trust in statements. Static analysis tools are known to give false positives and negatives; when a human analyzes an app by hand they sometimes miss things; and not all analysts are as good as others. For example suppose Alice is an experienced malware analyst: nine times out of ten if Alice says something is dangerous then it is. Bob, on the other hand, is a lazy analyst and doesn't always make the right call. His success rate is only 60%. It would be nice to formalize these levels of trust and express them in an authorization language. We could use these trust levels to state how confident we are with a decision when it was made on the basis of several judgements from variably trusted entities.

One approach to implementing this would be to attack the confidence to a SecPAL assertion as metadata. This would not modify how evaluation works and should not affect any of the safety proofs. Queries could be made based on the current confidence of all assertions by treating the confidence as a constraint. This is shown in Figure 34.

Levels of trust are interesting because they closer model how we make decisions in real life. Most static analysis tools are not utterly trustworthy. Being able model decisions based on levels of trust and quantify any statement allows us to model more closely the decisions we make and may allow us to express new, interesting policies.

6.5. Evaluation of App Guarden

The ideas presented in this project will be evaluated using the app store framework described in Section 6.2. The app store environment is ideal for testing with as we can measure the effects of policies on apps by grouping the apps into sets and seeing how they are filtered or modified. Methods for evaluating the success of the project for some ideas presented in this report are detailed bellow.

- To evaluate how users might use the policy language and their comprehension of policies we will let them use the app store. This will be presented to them through the use of a special *App Guarden app* that connects to the store and allows the download of apps based on policies. This will allow users to give us feedback and for us to ask questions that test how users might understand policies. Such questions should include:

Given this policy can you explain in natural language what it prevents? This tests the *comprehension* of a policy. If a user can describe what a policy is allowing then they may understand why a policy has prevented them from doing something.

Given this policy to do X can you modify it so that it instead does Y? This tests the *reconfigurability* of the policy language. This is important as (technically oriented) users will doubtless need to debug policies. A language that cannot be easily debugged will not be useful.

Here is an app. Do you expect this app would be allowed on your system given your policy? This again tests comprehension of policies but also tests how users relate these policies back to apps. If a user routinely expects an app to meet a policy, whereas it actually fails to meet it then this indicates that the user either does not understand their policy or does not understand how apps work. This may lead to users becoming frustrated and should be avoided.

You have asked us to install this app but we believe it fails to meet the policy P because of reason X. Do you really want to install it and why? This question allows us to judge how users *interact* with policies. If users are installing apps regardless of the warning this suggests the system is too cumbersome to use. If they are installing it because they believe the reasoning is wrong then this suggests either they understand the justification (if it is actually a bad reason) or they do not understand it (if the justification is correct). Users might override the system because they believe the policy is too draconian.

- To test the idea that apps can be filtered by policies we will write policies that describe different sets of apps; for example: games, flashlights, password safes and malware.

  Using the app store framework we will filter apps based on these policies. We will compare the apps the filters allow with their categorization from various different app stores. If a filtering policy for games allows an app the Play Store categorizes as a game we will consider it as a success with respect to the Play Store. If a filtering policy doesn't allow an app categorized as a game, or the policy allows an app that was not categorized as a game then we will consider it a failure with respect to the Play Store. When measured over a set of apps if the ratio of successful to unsuccessful filters will give a metric of how closely the filtering policy matches the app store categorization policy.

  By repeating with different app stores we can build different policies that closely match the different app store categorization policies. This then allows comparison based on any differences (or similarities) in the respective filter policies.

- To test the user policies we will start by creating policies for users that describe the restrictions we can imagine them wanting. From here we will attempt to write malicious[15] apps which deliberately break the policies. If these policies ensure that any malicious apps are kept away from the policy users (or at least flagged for modification) then the policy is successful. Extraneous malicious apps that get flagged by a policy may need further inspection to discover why, and any malicious apps that are allowed should count as policy failures. This allows us to measure the effectiveness of policies.

If policies require the use of various static analysis tools we can also assess the effectiveness of various configurations of the tools (or alternative tools which check the same things). To do this we write different policies which use different tooling set-ups and again measure the success rate on filtering apps from a malicious data set.

- To test how policies can prevent colluding apps we can take colluding apps (from app stores and synthetic examples); then measure how many of these apps a policy prevents. This also allows us to gain information about what kinds of collusion are most common: common collusions will result in more blocks. If we find we frequently block too many app then it suggests that our policies are too general, if we fail to block many apps then our policies are ineffective. If we cannot give understandable reasons why an app should be blocked then our explanations will need further work.

6.6. Conclusion

This proposal describes the aims and goals for my PhD. It describes several problems that need answering, and makes a case that researching how authorization logics can be used in mobile devices is interesting, non-trivial and may offer some worthwhile solutions to problems related to mobile devices.

The proposal provides a review of Android security tools, and the mechanisms already present in the systems. It describes existing logics of authorization, including those used for access control, and describes Datalog which is frequently used in their implementation.

My thesis is that logics of authorization can be applied to mobile devices; and that, by applying policies, we can improve device security. This will be accomplished by taking the decision making procedure from the users and instead enforcing their policies; instead of asking them to make a case-by-case decision.

The work will include showing a logic of authorization can model current security decisions, creating an app store framework that empowers users to describe their policies and have them enforced for them, and then creating new policies that tackle problems specific to mobile devices that existing authorization logics have not looked at and which do not have easy answers.

By describing how decisions are made within the Android ecosystem we will gain insight into what the trust decisions and authorization reasons are within the operating

[15]Malicious here may include traditional malware, but may also include PUS and handwritten apps that definitely break user's policies.

system. By expressing it in a formal language we can make precise comparisons between Android and other mobile operating systems. This formalization of the security policies in one language is novel and the increased understanding of the policy may help us understand its strengths and weaknesses better.

By using policies to describe apps we will be able to better understand how apps differ; even when the apps themselves are in similar categories. We will be able to understand how users decide what to install and what their expectations are with respect to security.

Mobile systems have specific problems. Existing uses of authorization logic have focused on systems in general, such as *Binder* or *SPKI/SDSI*, or on very specific cases (such as access control) within a system. Mobile devices have problems, such as collusion; app permissions; and personal data, that require greater attention than in other settings. This research aims to give a greater insight into how these problems arise and the trust relationships required to make the *right decision*.

This project is important because it allows us to improve our understanding of the decisions made in a mobile ecosystem. This will lead to better security for users as it allows policies to be written that match what they expect from an app and a device . The project will allow users to describe how apps should behave and ensure that they only use apps which meet these expectations. These user policies will allow for a consistent enforcement of policy and mean fewer decisions for the user. These policies allow precise comparison of different systems; this will allow an accurate comparison of security between them.

By the end of the PhD I want to have shown how authorization logics can be extended to describe and mitigate decision making problems on mobile devices. A schedule for completing the project is shown in Figure 35.

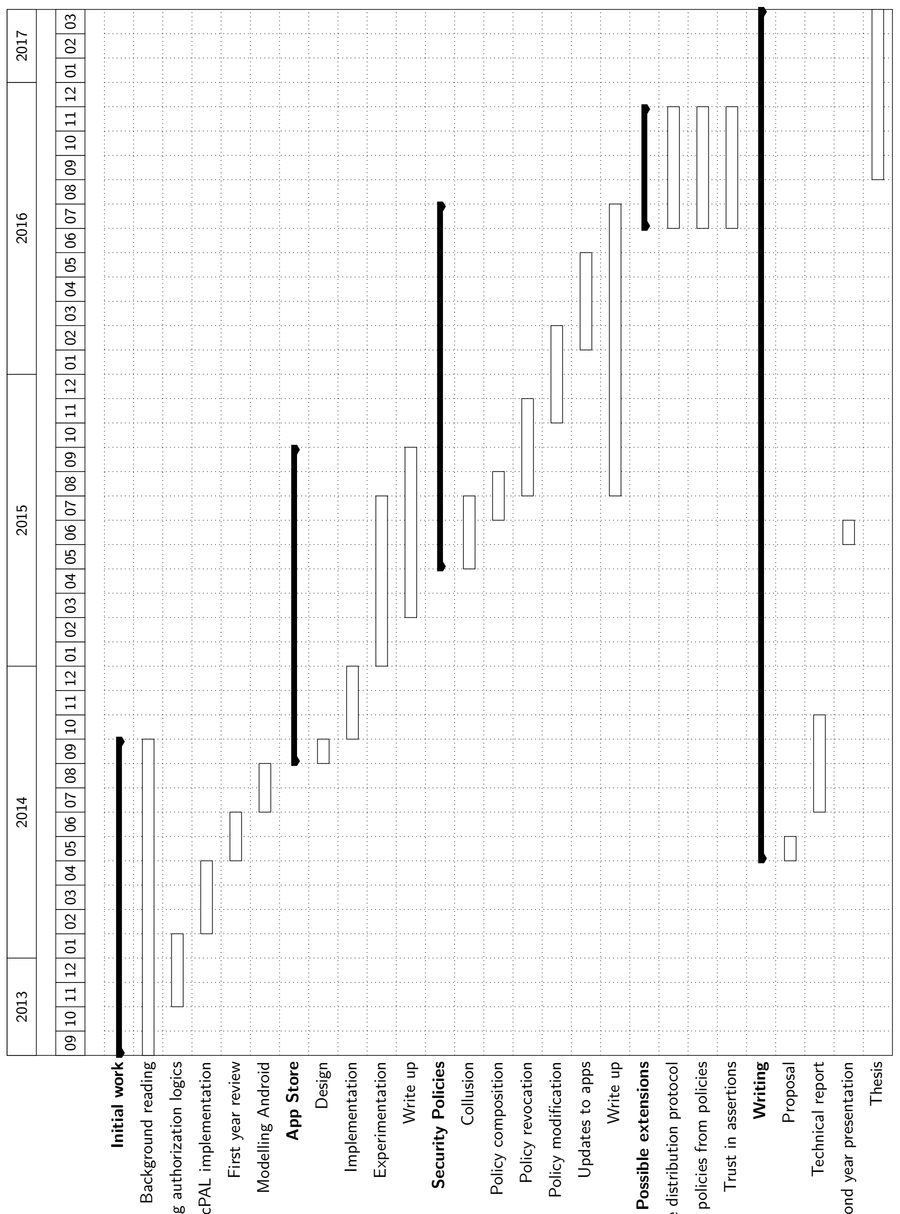


Figure 35: Gantt chart showing progress plans throughout funded period of PhD study.

A. References

[1] M Abadi. Logic in access control. In *Logic in Computer Science*, pages 228–233. IEEE Comput. Soc, 2003.

[2] A W Appel and E W Felten. *Proof-carrying authentication.* ACM, New York, New York, USA, November 1999.

[3] K R Apt, H A Blair, and A Walker. Towards a theory of declarative knowledge, 1986.

[4] B Aquilino, K Aquino, C Bejerasco, E Cajucom, S G Goh, A Hilyati, M Hyykoski, T Hirvonen, M Hypponen, S Jamaludin, C H Lim, Z Ong, M Suominen, S Sullivan, M Thure, and J Ylipekkala. All About Android. Technical report, F-Secure Labs, 2013.

[5] C Arthur. Walled gardens look rosy for Facebook, Apple – and would-be censors. *theguardian.com*, April 2012.

[6] K WY Au, Y F Zhou, Z Huang, and D Lie. PScout: analyzing the Android permission specification. *Computer and Communications Security*, pages 217–228, October 2012.

[7] M Backes, S Gerling, C Hammer, and M Maffei. AppGuard–Enforcing User Requirements on Android Apps. *Tools and Algorithms for the Construction and Analysis of Systems*, 2013.

[8] B Bajarin. Why Competing with Apple Is So Difficult — TIME.com. *TIME*, July 2011.

[9] F Bancilhon, D Maier, Y Sagiv, and J D Ullman. Magic sets and other strange ways to implement logic programs. *Special Interest Group on Management of Data*, pages 1–15, June 1985.

[10] M Y Becker. Secpal formalization and extensions. Technical report, Microsoft Research, 2009.

[11] M Y Becker and P Sewell. Cassandra: flexible trust management, applied to electronic health records. *Computer Security Foundations*, pages 139–154, 2004.

[12] M Y Becker, C Fournet, and A D Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Computer Security Foundations*, 2006.

[13] M Y Becker, A Malkis, and L Bussard. A framework for privacy preferences and data-handling policies. Technical report, Microsoft Research, 2009.

[14] M Blaze, J Feigenbaum, and J Lacy. Decentralized trust management. *Security and Privacy*, 1996.

[15] M Blaze, J Feigenbaum, and M Strauss. Compliance checking in the PolicyMaker trust management system. *Financial Cryptography*, 1465(Chapter 20):254–274, 1998.

[16] M Blaze, J Feigenbaum, and Angelos D Keromytis. KeyNote: Trust Management for Public-Key Infrastructures. *International Workshop on Security Protocols*, 1550 (Chapter 9):59–63, January 1999.

[17] M Blaze, Angelos D Keromytis, J Feigenbaum, and J Ioannidis. RFC 2704: The KeyNote Trust-Management System Version 2. Technical report, Network Working Group, 1999.

[18] J Blythe, R Koppel, and S W Smith. Circumvention of Security: Good Users Do Bad Things. *Security and Privacy*, 11(5):80–83, 2013.

[19] S Bratus, M E Locasto, M L Patterson, L Sassaman, and A Shubina. Exploit Programming. *login*, 36(6), December 2011.

[20] S Bugiel, L Davi, and A Dmitrienko. Towards taming privilege-escalation attacks on Android. *Network and Distributed System Security Symposium*, 2012.

[21] S Ceri, G Gottlob, and L Tanca. What you always wanted to know about Datalog (and never dared to ask). *Transactions on Knowledge and Data Engineering*, 1(1): 146–166, 1989.

[22] E Chien. Motivations of recent android malware. *Symantec Security Response*, 2011.

[23] J Chomicki, D Goldin, G Kuper, and D Toman. Variable independence in constraint databases. *Transactions on Knowledge and Data Engineering*, 2000.

[24] N Damianou, N Dulay, E Lupu, and M Sloman. The Ponder Policy Specification Language. *Policies for Distributed Systems and Networks*, pages 18–38, January 2001.

[25] J DeTreville. Binder, a logic-based security language. In *Security and Privacy*, pages 105–113. IEEE Comput. Soc, 2002.

[26] Joshua J Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A Ridley, and Georg Wicherski. *Android Hacker's Handbook*. John Wiley & Sons, March 2014.

[27] T Eastep. Shorewall. *Shorewall*.

[28] C Ellison, B Frantz, B Lainpson, R Rivest, and B Thomas. *RFC 2693: SPKI certificate theory*. The Internet Society, 1999.

[29] W Enck, M Ongtang, and P McDaniel. On lightweight mobile phone application certification. *Computer and Communications Security*, pages 235–245, November 2009.

[30] W Enck, P Gilbert, B G Chun, L P Cox, and J Jung. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *Operating Systems Design and Implementation*, 2010.

[31] A P Felt, E Chin, S Hanna, D Song, and D Wagner. Android permissions demystified. *Computer and Communications Security*, pages 627–638, October 2011.

[32] A P Felt, E Ha, S Egelman, A Haney, E Chin, and D Wagner. Android permissions: user attention, comprehension, and behavior. *Symposium On Usable Privacy and Security*, page 3, July 2012.

[33] C Fritz, S Arzt, and S Rasthofer. Highly precise taint analysis for android applications. Technical report, Technische Univerität Darmstadt, 2013.

[34] A P Fuchs, A Chaudhuri, and J S Foster. SCanDroid: Automated security certification of Android applications. *USENIX Security Symposium*, 2009.

[35] P Gama and P Ferreira. Obligation Policies: An Enforcement Platform. *Policies for Distributed Systems and Networks*, 2005.

[36] Michael Gelfond and Jorge Lobo. Authorization and Obligation Policies in Dynamic Systems. *International Conference on Logic Programming*, pages 22–36, January 2008.

[37] Google. Android SEPolicy: `https://android.googlesource.com/platform/external/sepolicy/`.

[38] Y Gurevich and I Neeman. DKAL: Distributed-Knowledge Authorization Language. *Computer Security Foundations*, pages 149–162, 2008.

[39] J Hallett and D Aspinall. Towards an authorization framework for app security checking. In *ESSoS Doctoral Symposium*. University of Edinburgh, February 2014.

[40] P Hornyack, S Han, J Jung, and S Schechter. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Computer and Communications Security*, 2011.

[41] J Jeon, K K Micinski, J A Vaughan, A Fogel, N Reddy, J S Foster, and T Millstein. Dr. Android and Mr. Hide: fine-grained permissions in android applications. *Security and Privacy in Smartphones and Mobile Devices*, pages 3–14, October 2012.

[42] L Kagal and A Rei. Rei: A Policy Language for the Me-Centric Project. Technical report, HP Labs, 2002.

[43] J Kruger and D Dunning. Unskilled and unaware of it: How difficulties in recognizing one's own incompetence lead to inflated self-assessments. *Journal of Personality and Social Psychology*, 77(6):1121–1134, December 1999.

[44] B Lampson, M Abadi, and M Burrows. Authentication in distributed systems: Theory and practice. *Special Interest Group on Operating Systems*, 1992.

[45] N Li and J C Mitchell. Design of a role-based trust-management framework. *Security and Privacy*, 2002.

[46] N Li and J C Mitchell. Datalog With Constraints. *Practical Aspects of Declarative Languages*, 2562(Chapter 6):58–73, January 2003.

[47] N Li, W H Winsborough, and J C Mitchell. Distributed credential chain discovery in trust management. *Journal of computer security*, 2003.

[48] B Livshits, M Sridharan, Y Smaragdakis, O Lhoták, J N Amaral, B-Y E Chang, S Guyer, U Khedker, A Møller, and D Vardoulakis. In Defence of Soundiness: A Manifesto. Technical report, Soundiness, 2014.

[49] A Moulu. From 0 perm app to INSTALL_PACKAGES on Samsung Galaxy S3, July 2012. URL `http://sh4ka.fr/android/galaxys3/from_0perm_to_INSTALL_PACKAGES_on_galaxy_S3.html`.

[50] G C Necula and P Lee. Proof-carrying Code. Technical Report CMU-CS-96-165, Carniegie Mellon University, 1996.

[51] S Rasthofer, S Arzt, and E Bodden. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. *Network and Distributed System Security Symposium*, 2014.

[52] J Rutkowska and R Wojtczuk. Qubes OS architecture. Technical report, Invisible Things Lab, 2010.

[53] G Sarwar, O Mehani, and R Boreli. On the effectiveness of dynamic Taint analysis for protecting against private information leaks on android-based devices. *International Conference on Security and Cryptography*, 2013.

[54] B Schneier. "Schneier's Law", April 2011. URL `https://www.schneier.com/blog/archives/2011/04/schneiers_law.html`.

[55] C Shaufler. v8 Simplified Mandatory Access Control Kernel. *Linux Security Module Mailing List*, July 2007.

[56] R Spencer, S Smalley, P Loscocco, and M Hibler. The Flask security architecture: System support for diverse policies. *USENIX Security Symposium*, 1999.

[57] I Stark. Reasons to Believe: Digital Evidence to Guarantee Trustworthy Mobile Code. In *The European FET Conference*, September 2009.

[58] V Svajcer. When Malware Goes Mobile: Causes, Outcomes and Cures. Technical report, Sophos, October 2012.

[59] V Svajcer and S McDonald. Classifying PUAs in the Mobile Environment. *sophos.com*, October 2013.

[60] A Tongaonkar, N Inamdar, and R Sekar. Inferring Higher Level Policies from Firewall Rules. *Large Installation System Administration Conference*, 2007.

[61] K Twidle, N Dulay, E Lupu, and M Sloman. *Ponder2: A Policy System for Autonomous Pervasive Environments.* IEEE, 2009.

[62] United States Government Department of Defence. *Trusted Computer System Evaluation Criteria.* United States Government Department of Defence, United States Government Department of Defence, 1985.

[63] A Uszok, J Bradshaw, R Jeffers, N Suri, P Hayes, M Breedy, L Bunch, M Johnson, S Kulkarni, and J Lott. KAoS policy and domain services: toward a description-logic approach to policy representation, deconfliction, and enforcement. *Policies for Distributed Systems and Networks*, pages 93–96, 2003.

[64] T Vidas, N Christin, and L Cranor. Curbing android permission creep. In *Proceedings of the Web*, 2011.

[65] N Whitehead, M Abadi, and G Necula. By reason and authority: a system for authorization of proof-carrying code. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 236–250. IEEE, 2004.

[66] E Wobber, M Abadi, M Burrows, and B Lampson. Authentication in the Taos operating system. *Transactions on Computer Systems*, 12(1):3–32, 1994.

[67] M Zheng, M Sun, and JCS Lui. DroidRay: A Security Evaluation System for Customized Android Firmwares. *ASIA Computer and Communications Security*, 2014.

[68] Y Zhou and X Jiang. Dissecting Android Malware: Characterization and Evolution. *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109, 2012.