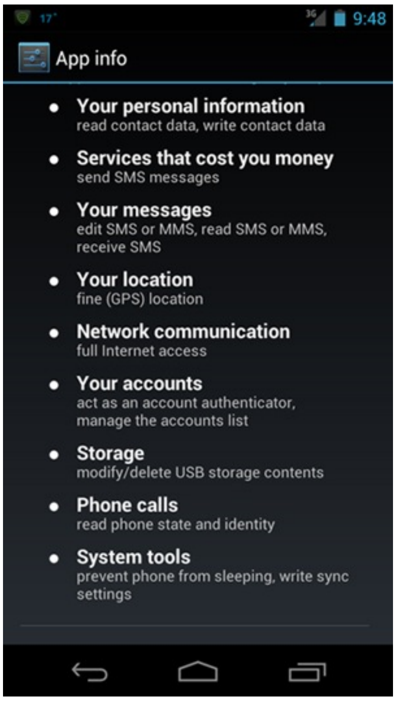
Using Authorization Logics To Model Security Decisions in Mobile Systems

Thesis Proposal

Joseph Hallett

June 11, 2014

Contents



- **Your personal information**
  read contact data, write contact data
- **Services that cost you money**
  send SMS messages
- **Your messages**
  edit SMS or MMS, read SMS or MMS, receive SMS
- **Your location**
  fine (GPS) location
- **Network communication**
  full Internet access
- **Your accounts**
  act as an account authenticator, manage the accounts list
- **Storage**
  modify/delete USB storage contents
- **Phone calls**
  read phone state and identity
- **System tools**
  prevent phone from sleeping, write sync settings

Figure 1: Some of the 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 [28]. 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 a large number of 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 [27]: they request permissions that grants them access to data they do not use. Some apps are malicious [53]: 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 battery [46].

More generally users and computers make decisions. Whether it is to update an app or to

[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.

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* [23, 47] 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*. 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, detect malware, static 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* [11] 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 [40] 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[31]. 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 someone 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 14); 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 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[10], 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 [11], 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

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

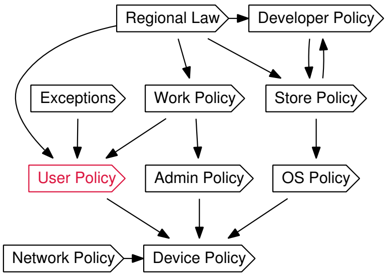


Figure 2: Different policies being applied to one another. Users, businesses and stores are all subject to regional laws. A store may have a policy but the developers who write the apps may also add their own rules in. Devices might have their own policies set by their designers but also have the OS 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.

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 called their *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. An example of how a compositional policy might be written is shown in Figure 1.2.

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?

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

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

Phone says User can-say inf
  app meets UserSecurityPolicy.

Phone says PlayStore can-say 0
  app meets AppStorePolicy.

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

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

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 [27, 49] 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.

1.3 Personally 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 to a security policy we also

[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	Worldwide	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	Worldwide	40×10^3	15×10^3	Using multiple malware scanners including one by *Blue-Box security* that can detect apps exploiting the master key vulnerability.	Installed by 140 OEMs. Twenty million users.
Amazon AppStore	Worldwide	76×10^3	25×10^6		Used on Kindle tablets, but popular on Android.

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

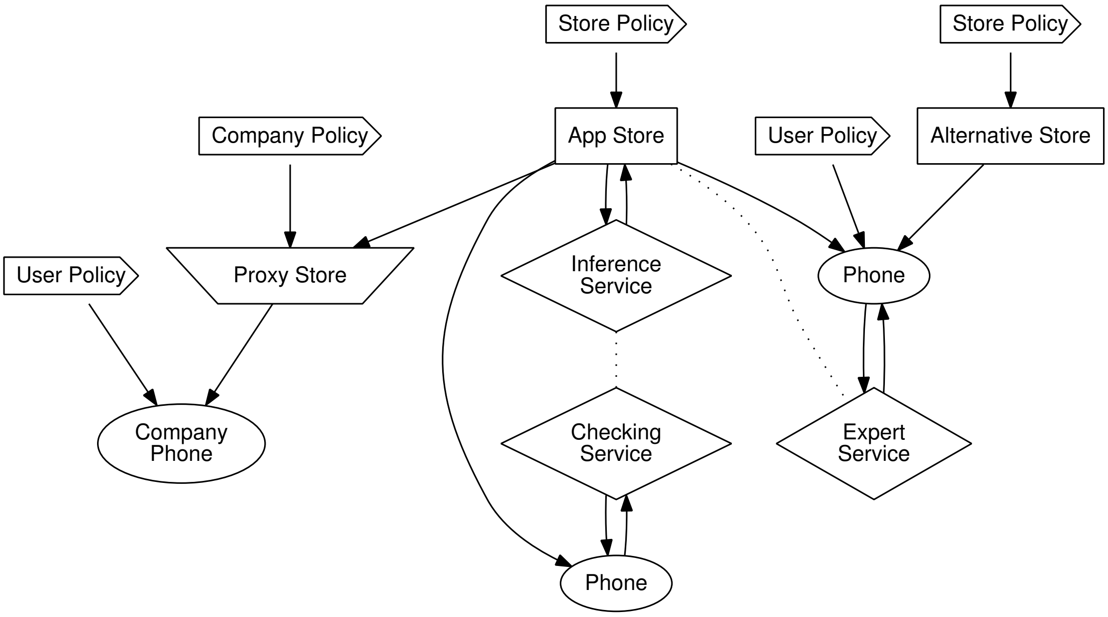


Figure 5: Overview of the different entities in App Guarden. Each of the ovals represents a different device inside the system; each rectangle or trapezium represents different supplier of apps, and each diamond is an authority that makes statements about apps and other software decisions. Arrow boxes show policies. Arrows represent transfers of information or apps, and dotted line indicate that the two entities are connected in some other way.

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

To enhance trust in the store digital evidence could be offered with the apps This would give devices a practical means to check the app is supported by their security policy without having to re-run all the static analysis checks themselves. This should also save device battery life.

Proof-carrying authentication [2] and authorization logics such as BLF [50] 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.

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

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

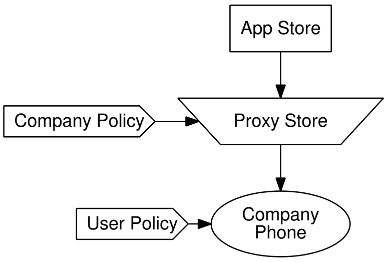


Figure 6: Security policies and the proxying store

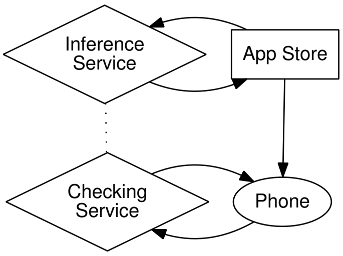


Figure 7: Checking services and an app store.

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 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.

Another example might be an app store that supplies apps with *digital evidence*. When the app store sells an app it wants to reassure its users that it real guarantees that the app it is selling meets the security guarantees 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.

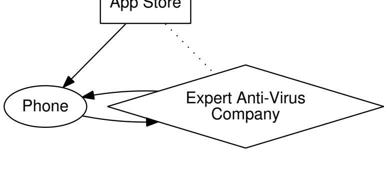


Figure 8: Use of an expert checker.

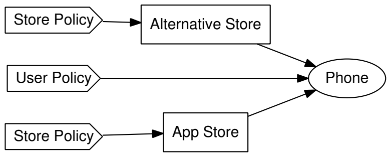


Figure 9: A device using multiple stores with different policies.

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.

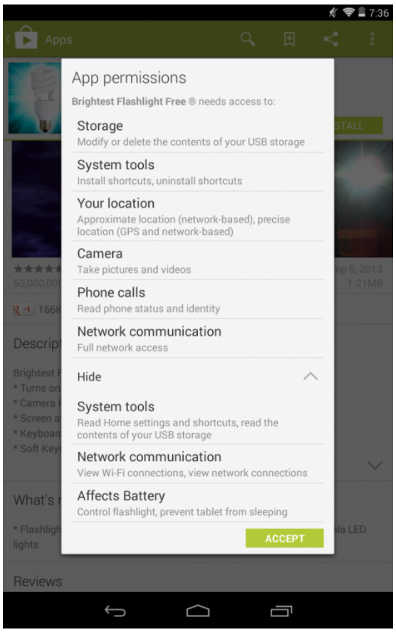
2 Review of Android Security

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 [22].

2.1 Permissions and 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 access. For example the `INTERNET` permission allows apps to access the network. To enforce this Android uses the Linux file system permissions in the underlying 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



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

ACCEPT

Figure 10: 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.

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 permissions [52].

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 [28]. 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 [19]) 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 10). The *Stowaway* tool [27] 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 [5] 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

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

dependant typing.

The *RefineDroid, Dr. Android & Mr. Hide* tools [34] 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 [33] doesn't modify apps. Users can write policies for what data an app can receive. If an app breaks this then the it is stopped or fake data supplied instead. This requires changes to Android however. The *AppGuard* tool [6, 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.

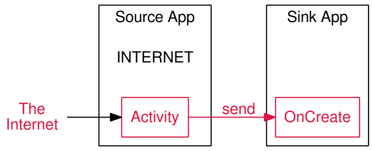


Figure 12: A flow between components a tool like SCanDroid might catch. The aim would be to detect that data from the internet is send to an activity app which can then be sent to an app without the internet permission.

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* [25] 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
                        ]
```

2.2 Intents and Collusion

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 [39].

Tools have been made to find privilege escalation attacks. *Quire* [17] added origin tracing to intents. *SCanDroid* [30] statically analyzed apps to find flows across components. It describes constraints that should be satisfied to stop leaks.

TaintDroid [26] and *FlowDroid* [29] 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 [42]: it can be defeated by malicious apps. FlowDroid takes a list of sources and sinks (found using the *SuSi* tool[41]) 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 13: Example usage of the PolicyMaker authorization language. The policy block says that only key 0x12345678 can include in their message that they are Alice at Microsoft. The first request is from the key 0x12345678. It would be authorized as the regular expression matches. The second request is from a different key. It would not be authorized by the policy.

3 Review of Policy Languages

3.1 Logics of Authorization

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* for 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 [13], grew out of the logics of authentication proposed by Wobber, Abadi, Burrows, and Lampson [35] [51]. PolicyMaker allows other principals (identified through asymmetric keys) to be trusted for actions or to declare further relationships. The language was minimal. It did not specify how the policies should be checked: they suggested regular expressions, or a special version of AWK. Any language could have been 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 3.1. If we received a request from a different key to say they’re Alice it would be denied; wheres a message from Alice’s key would be authorized.

Checking whether a PolicyMaker policy is satisfied is NP-hard [14]. It is not tractable as checking complicated policies can take exponential time. PolicyMaker allows arbitrary checking programs to be used in assertions. Deciding whether a program will stop when given an arbitrary 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 either. 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 [15] which was a revised version of PolicyMaker for public-key infrastructure. Like PolicyMaker it authorized actions based on keys and a series of conditions. It dropped support

```
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 14: The policy from Figure 3.1 rewritten in KeyNote.

for arbitrary program checkers; opting for its own specific language [16]. An example of KeyNote is given in Figure 14.

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* [24] was more complex. KeyNote even claimed 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 authorize anyone working with them to be able 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 [36] built on PolicyMaker. RT allowed principals to be given roles; similar to a Role-Based Access Control (RBAC) system. Decisions were made based on which roles were 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:

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

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

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

Figure 15: Role delegation in the *Cassandra* policy language. A manager is allowed to activate the employee role for an arbitrary entity by appointing them.

> “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 [38] and with RT_1 and RT_2 adding support for parameterized-roles and logical-objects respectively. Extensions added support for constraints. This allowed RT_1^C[37] 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} *[37]*). Datalog is known to be tractable. 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.

Cassandra [10] 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.

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 [21] 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.

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

Figure 16: 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.

3.2 SecPAL

SecPAL [11] is an authorization logic for decentralized systems. Early experiments indicate that it is good for modeling the distributed nature of software installation, app stores and mobile devices. We will describe it in more detail than other languages.

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. 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 inf x is-allowed.
B says C can-say inf x is-allowed.
C says D can-say inf 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).

```
User says SMSSender can-send-message-to(m)
  if m is-in-addressbook.
User says ContactsApp can-say 0
  person is-in-addressbook.

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

Figure 17: Use of SecPAL's *can-act-as* statement to apply restrictions from one messaging app to another.

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

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

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

Figure 18: 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 ∞.

```
// Non-standard SecPAL
A says B can-say inf
  c can-say inf
    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 were to 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 17.

Extensions of SecPAL [9] add support for guarded universal quantification. They also remove the *can-act-as* statement. Other languages such as *DKAL* [31] built on and eventually split from SecPAL. DKAL was designed to express distributed knowledge between 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 [12] 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 18. 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. This safety conditions ensures that

```
e ::= x  // variables
    | A  // constants
pred ::= ''possesses''  // predicates
       | ''can''
       | ...
D ::= 0      // no delegation
    | ''∞'' // delegation
verbphrase ::= pred e_1 ... e_n
             | ''can say'' D fact
             | ''can act as'' e
fact ::= e verbphrase
claim ::= fact ''if'' fact_1, ..., fact_n, c
assertion ::= A ''says'' claim
AC ::= assertion_1...n // assertion context
```

Figure 19: BNF specification of the SecPAL language.

when the SecPAL program is translated into Datalog all constraints become ground (they do not contain variables). This ensures they're easy to solve.

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 * can-say * *
            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, *):
    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 $\wedge$ condition-2 $\wedge$ condition-3

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

        condition-2 = Vars(c) $\subseteq$ Vars($vFs$)
        condition-3 = $\forall$ f $\in$ fs :
            is-flat(f)
```

3.3 Access Control Systems

Access control is an area where logics of authorization have been successfully applied. 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*[48] for all but the least secure systems.

SELinux is based on the Flask security architecture[44]. 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 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[43], 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 20.

Role based schemes improve over MAC ones 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

```
C         Unclass rx
S         C       rx
S         Unclass rx
TS        S       rx
TS        C       rx
TS        Unclass rx
```

Figure 20: A hierarchical security policy for the SMACK access control system. Top secret (TS) can read secret (S), classified (C) and unclassified (unclas) documents; secret can read classified and unclassified but cannot read secret documents and so on.

are not defined for any specific user. This means that a 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 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 a database language. It was created from a simplification of general logic programming. The language is based on first order logic; it is both sound and complete. 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.

[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.

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

Datalog programs are presented as series of Horn clauses in the syntactically same way as the Prolog language (see Figure 4). There are additional restrictions, however, that all variables in the head of a clause must be present in the body; and that 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 [18]. Given a Datalog program try every constant with every rule from the IDB. When a rule is found to be true 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 it stops then it is false; as Datalog assumes the closed world assumption (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* [8] 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. The Prolog language (which Datalog is a more constrained form of) uses this strategy. Its use of 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. Tabling is often used with this strategy to speed queries by memoizing previously inferred facts.

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 possible to construct queries which return an infinite number of answers.

4.2 Datalog Evaluation in SecPAL

The bottom up strategy is commonly used with Datalog programs. Becker's paper describing SecPAL [11] points out that since their programs may change dramatically for every query 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.

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.

To 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 follows:

```
# 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 $\exists$ q' $\in$ prog.ans — can-rename($q_0$, q') :
                            wait(q') $\xleftarrow{+}$ nd
                            for nd' $\in$ wait(q') :
                                if $\exists$ nd'' = resolve(nd, nd') :
                                    process-node(nd'')
                        else
                            wait($q_0$) $\xleftarrow{+}$ 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 $\notin$ ans(p) :
                ans(p) += nd
            for nd' $\in$ wait(p) :
                if $\exists$ nd'' = resolve(nd', nd) :
                    process-node(nd'')
```

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 result: for example an app is safe to run if it is not malware.

A version Datalog with negation called *Datalog*$^\neg$ [18] 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 [37]) is based on constraint logic programming. Constraint logic programming allows relationships to be defined with general relationships (for example: less than $<$) rather than with just 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 [10], SecPAL [11] and RT$_1^C$ [37] 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 [20] 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 Proposal

5.1 Work Done In First Year

The first year of my study has been on developing an authorization logic that can express the security policies for a smart phone. Specifically the policies when a user is installing apps. We have considered what kinds of policies and trust relationships a user might wish to express and shown how they can be expressed in the language.

To do this we initially looked at a variety of authorization logics. These included BLF [50] and Binder [21]. We settled on SecPAL as it was 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 wanted 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 [45] 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 wanted 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. The *meets* predicate says an entity believes an app meets an application policy. If Alice believed the *Angry-Birds* app met her policy to not leak information about her:

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

To express proof carrying code [40] 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.
```

5.2 Alice Installs An App

To illustrate we describe a story where a user is trying to install an app. This example is built from work presented as a paper at the ESSoS Doctoral Symposium [32], and as a poster at the FMATS workshop.

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 wont 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 inf app meets NotMalware.
  Alice says NLLTool can-say 0 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 22. Alice uses SecPAL to decide whether it says that `Alice says app is-installable.`

```
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 inf
  app meets NotMalware.
Alice says NLLTool can-say 0
  app meets NoLocationLeaks.
Google says McAfee can-say 0
  app meets NotMalware.
McAfee says
  AngryBirds meets NotMalware.
NLLTool says ABProof shows
  AngryBirds meets NoLocationLeaks.
```

Figure 22: The full assertion context used to evaluate Alice's query.

5.3 Implementation

We have implemented the SecPAL logic. The implementation was done in Haskell and is around a thousand lines of code, plus five hundred lines of test cases.

In the original SecPAL paper [11] 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. It 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.

An example of a proof generated by the tool is shown in Figure 23. 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.

5.4 Thesis Proposal

A schedule for completing the project is shown in Figure 24.

I would like to focus on developing security policies for mobile systems. My first year has focussed on exploring SecPAL and ensuring it is the right logic to model the issues surrounding smart phones. The next two years will be spent exploring what happens when these policies interact with users.

First 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; that it is capable of describing complex security policies. This will result in a technical report. The report will describe the authorization logic, why it was chosen over other policy languages. I will show some applications of the logic to mobile systems and the problems associated with them.

Next I will develop an app store. The store will use security policies to filter apps. Creating

```
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 23: Proof output by the SecPAL tool when evaluating Alice's query.

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.

The store will act as a framework to compare different store's policies with. By checking policies in the app store it avoids needing for users to root their phone (reducing their device security). It also allows 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. The engineering difficulty in developing such a store can be mitigated by sensible software development practices.

Next we will increase the complexity of the policies and show how the policies interact with the user. One area will be on compositional policies; where a user might have one policy for apps at home and another for how they should use their phone at work. Showing that SecPAL could support policies of this kind is, as hinted earlier, easy; however it is not clear what to do when these policies change, or when new policies are composed with them, or when two composed policies contradict each other.

Another area will be to show how policies can be written to take into account of the *app collusion problem*. Whilst tools have been written 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. By developing an authorization logic to model these decisions will allow for a richer policy language for mobile devices.

The final 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

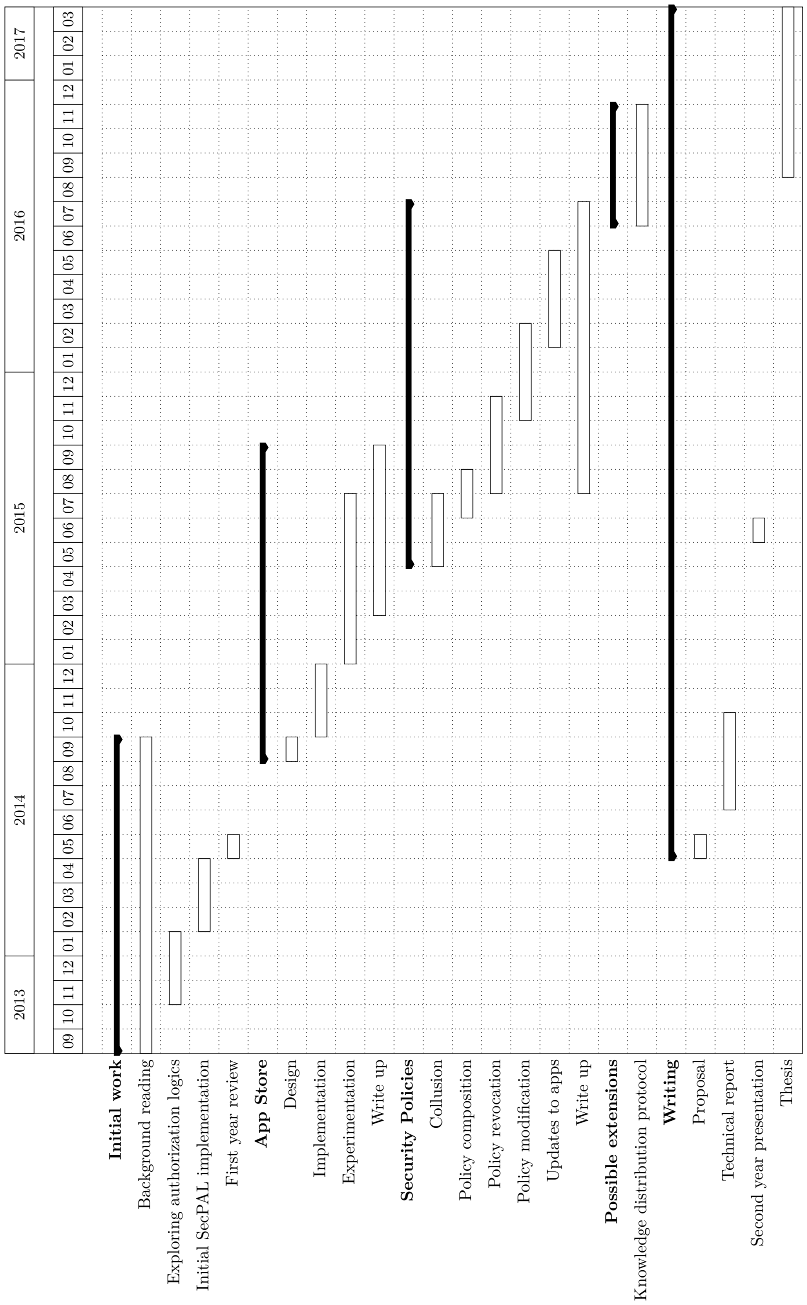


Figure 24: Gantt chart showing progress plans throughout funded period of PhD study.

well as developing policies that can describe what to do when a well-used app no longer meets the security policy.

At the end of the project I want to have shown how authorization logics can be extended to describe and mitigate decision making problems on mobile devices.

References

[1] M Abadi. Logic in access control. In *Logic in Computer Science, 2003. Proceedings. 18th Annual IEEE Symposium on*, 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] 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.

[6] M Backes, S Gerling, C Hammer, M Maffei, and P von Styp-Rekowsky. AppGuard - Real-time policy enforcement for third-party applications. Technical report, July 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] 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.

[9] M Y Becker. Secpal formalization and extensions. Technical report, 2009.

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

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

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

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

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

[15] 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.

[16] 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.

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

[18] 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.

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

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

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

[22] 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.

[23] T Eastep. Shorewall. *Shorewall*.

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

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

[26] 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.

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

[28] 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.

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

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

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

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

[33] 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.

[34] 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.

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

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

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

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

[39] 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`.

[40] G C Necula and P Lee. Proof-carrying Code. Carniegie Mellon University, 1996.

[41] 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.

[42] 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.

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

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

[45] I Stark. Reasons to Believe: Digital Evidence to Guarantee Trustworthy Mobile Code. In *The European FET Conference*, pages 1–17, September 2009.

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

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

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

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

[50] 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.

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

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

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