Authorization Logic for Mobile Ecosystems

Second Year Report

Joseph Hallett

September 1, 2015

This report describes the second year work of my PhD. I give a brief overview of app installation policies and the AppPAL authorisation logic; before describing the work completed this year. I review topics suggested in the thesis proposal; and describe my work implementing AppPAL, exploring store and user policies and looking at protocols for app distribution. I describe what I expect to work on in the final year of my studies; and, finally, I give a table of contents and plan for my producing my thesis in the remaining time available to me.

1. Introduction

Mobile devices are different to other computers. They have access to greater personal data than a PC; pictures and photos as well as conversations, locations, and address books. They mostly only run software sold through stores under the manufacturer's control. People take them to work and on holiday; into their homes and out to the towns and countryside. These devices communicate through wireless almost exclusively: working with cars via bluetooth, the internet via 3G, speakers TVs and other internet-of-things devices through wifi. These differences create a different ecosystem for them to operate in: one that we aim to model and control using authorization logic.

Authorization logics are designed to express policies which decide whether a principal, for example a computer user, can perform an action. They have been used to implement access control systems to decide whether users can read or write to files. I believe they can also be used to help users decide which apps they want to use.

The interaction between stores, users, apps and phones gives rise to trust relationships between each of the different entities. Users trust the stores to sell them good, malware-free apps. Some stores will sell any apps, but others are more picky relying on audits and tools to find the higher quality apps to sell. In the stores users are told about the apps they might want to buy (Figure 1): who developed the app, whether they were a *top developer*, what kind of app it is, age ratings, functionality the app may have, aggregated review scores, descriptions and bugs from both anonymous and not so anonymous users, update times, the price of in-app-purchases and the permissions it will request. How do the device owners know which information to trust? Where does the store get this information from in the first place?

Finding the right apps can be tricky: users need to work out which apps are well written, which are not going to abuse their data, which ones will work in the way they want. Users and companies need to find the apps which suit how they want to use their devices. This can be difficult as it isn't obvious how apps use the data they have access to. There is a large amount of choice. Apps are picked for a variety of reasons: ordering in the store [12], reviews and privacy concerns [9], and security rules from employers. These policies could be written down and used to guide device owners when they are making a decision that goes against them.

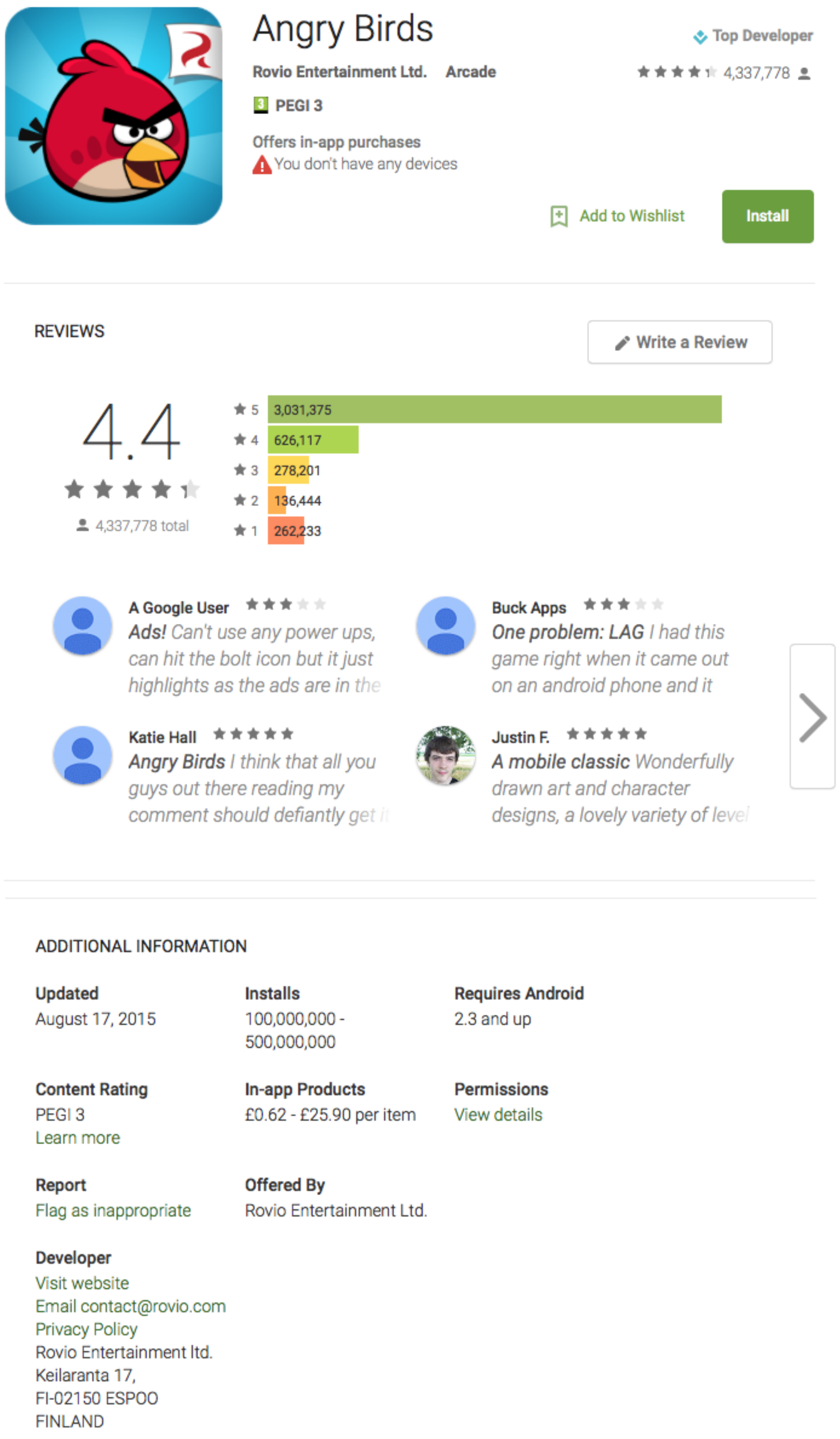
For example, one user may wish to only install games that have favourable reviews on their favourite gaming site, and which don't have in-app purchases. For this user searching a store might be frustrating. They need to work out the permissions of the app, cross reference the apps between the store and the review site. Using an authorization logic (like AppPAL) we could express the preferences as a policy and by enforcing it only show the user the apps they wanted in the store. The policy will need to delegate to the review site to decide if the app is good, and to google to decide the category of apps. We might also need to examine the app itself to decide whether it can make in-app purchases[1].

```
'user' says App isInstallable
  if App hasCategory('game'),
     App isGood
  where hasPermission(App, "IAP") = false.

'user' says 'play-store' can-say 0 App hasCategory(Category).
'user' says 'review-site' can-say inf App isGood.

'review-site' says App isGood
  where reviewSiteScore(App) > 7.
```

If an employer uses a bring your own device (BYOD) scheme in their company they may trust users to follow company IT policies when using their phones at work. Using tools like Google's *Apps Device Policy*[2] the employer can restrict the use of specific apps. This is an unsubtle form of control. We want to block apps by describing the characteristics of the apps we want or do not want. Blacklisting requires constant updating to keep the list up-to-date. Blacklisting doesn't allow for rich policies using information from third parties or on the basis of metadata about the app. There is no scope for making any kind of decision other than to allow an app or not. For IT departments in these companies these kinds of tools are the only tool to enforce their policies. For more advanced policies, such as not leaking company documents, compliance is often left to employees.

[1]On Android the ability to make in-app purchase is controlled through the permissions system.

[2]`https://play.google.com/store/apps/details?id=com.google.android.apps.enterprise.dmagent`



Angry Birds

Top Developer

Rovio Entertainment Ltd. Arcade

★★★★½ 4,337,778

PEGI 3

Offers in-app purchases

You don't have any devices

Add to Wishlist

Install

REVIEWS

Write a Review

4.4

★★★★½

4,337,778 total

★ 5	3,031,375
★ 4	626,117
★ 3	278,201
★ 2	136,444
★ 1	262,233

A Google User ★★★☆☆
Ads! Can't use any power ups, can hit the bolt icon but it just highlights as the ads are in the

Buck Apps ★★★☆☆
One problem: LAG I had this game right when it came out on an android phone and it

Katie Hall ★★★★★
Angry Birds I think that all you guys out there reading my comment should defiantly get it

Justin F. ★★★★★
A mobile classic Wonderfully drawn art and character designs, a lovely variety of level

ADDITIONAL INFORMATION

Updated
August 17, 2015

Installs
100,000,000 - 500,000,000

Requires Android
2.3 and up

Content Rating
PEGI 3
Learn more

In-app Products
£0.62 - £25.90 per item

Permissions
View details

Report
Flag as inappropriate

Offered By
Rovio Entertainment Ltd.

Developer
Visit website
Email contact@rovio.com
Privacy Policy
Rovio Entertainment ltd.
Keilaranta 17,
FI-02150 ESPOO
FINLAND

Figure 1: Some of the information presented to the user in the Play store.

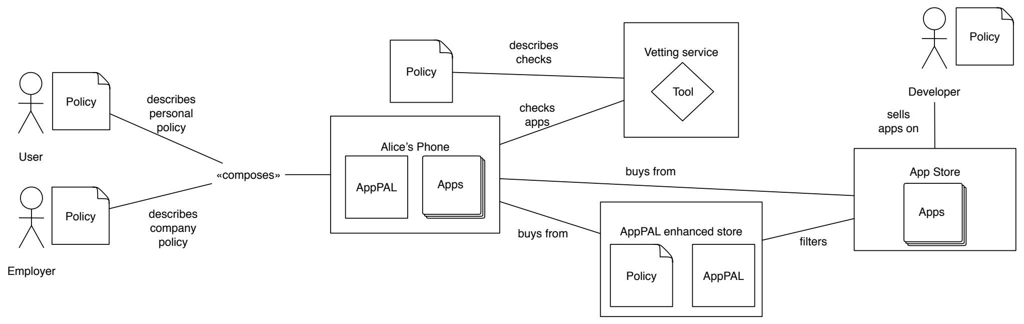


Figure 2: The mobile ecosystem surrounding a phone used for BYOD and the trust relationships surrounding it.

Policies exist for mobile ecosystems; but there aren't ways of precisely modelling or enforcing them. Authorization logic could be used to describe these policies and enforce them automatically. Logics can describe the trust relationships and policies surrounding Android formally. This lets us to make precise statements about the security of these systems. They can be used to make comparisons between different security models, such as those in iOS and Android. It allows comparisons between user's privacy policy and their actual behaviour.

By capturing these patterns as policies they can be enforced automatically. By checking the policies we can enforce them at run-time; or warn when making decisions that go against them. This reduces the burden on users to decide which apps they want. Security-savvy users may design policies themselves: these could be shared with others or used in organisation-wide curated app stores.

This thesis research will show how authorization logic can be used to make security decisions in mobile devices. Currently, security decisions must be made manually by smart phone users By automating these choices we believe users can avoid having to make security decisions and their overall security be improved.

1.1. AppPAL

AppPAL is an instantiation of Becker et al.'s SecPAL [3] I have developed to model and enforce app installation policies. Using AppPAL we can describe policies with trust relationships, and which incorporate constraints. This gives a powerful language to describe user's and company's app policies precisely.

Using AppPAL a principal (someone who utters statements about apps, such as a user or store) can say an app is installable.

```
'user' says 'apk://com.rovio.angrybirds' isInstallable.
```

Or that an app is installable only *if* some conditions are correct and *where* a constraint is true.

```
'user' says App isInstallable
  if 'office-app-policy' isMetBy(App)
  where locationAt('work') = true.
```

There is support for renaming or role assignment.

```
'user' says 'ian' can-act-as 'it-department'.
```

Statements are collected into an *assertion context* which is queried using SecPAL's three evaluation rules [3].

Decisions can be delegated to other principals; with separate versions where further delegation is allowed (*inf*) or not (*0*). For example as part of a company policy all apps should come with a statement from an anti-virus vendor that the app is not malicious. To create this relationship the *can-say* statement is used. For example they may trust McAfee and Google to decide whether something is malicious or not, but want McAfee to say directly whether an app is malware. With Google, however, they're happy for them to recommend another anti-virus vendor they trust to make the decision for them. In the case of McAfee we use *can-say0* and with Google *can-sayinf*.

```
'company' says 'mcafee' can-say 0 App isMalicious.
'company' says 'google' can-say inf App isMalicious.
```

If, when checking if an app is malicious, we come across a statement from McAfee delegating then the delegation will be ignored.

```
'mcafee' says 'f-secure' can-say 0 App isMalicious.
```

A delegation from Google, however, could be used to decide whether the app was malicious.

```
'google' says 'f-secure' can-say 0 App isMalicious.
```

2. Summary of second year work

I proposed spending this year looking at the role of app stores in my thesis proposal. Specifically I wanted to understand how users interacted with the stores. This would enable us to better understand their behaviour letting us write policies that accurately described their actions. I said that I would:

- Implement an app store filtered by AppPAL policies.
- Look at how users interact with apps, and the kinds of policies they apply, as well as what happens when their policies change.
- Look at how the policies vary within categories of apps.
- Explore how policies could be composed and joined.

- Explore what happens when policies are attacked.

These topics changed as the year went on. I implemented a system for creating app stores based on an AppPAL policy, but I also gained access to data saying what users installed. Rather than working on a system to let users filter apps and then analysing the policies they were using, I tried to look at what apps users pick from stores and the kinds of policies they apply using the current app stores. I started looking at greater detail at the distribution mechanisms and policies within app stores.

Some existing work continued on from the first year: AppPAL was reimplemented in Java to allow it to run on Android. Other work was started from necessity. I wanted a means of systematically running tools over large numbers of apps and collecting the results. So I built a security knowledge base (SKB) and have used the data collected from it in other parts of the project.

I describe work I have completed this year and show some findings bellow. My work this year has been exploring the policies surrounding the app stores. To do this I needed a means of storing information about the apps they sell, and a means of checking policies (preferably on a mobile device). From there I could look at the policies specified by the stores themselves, and implicit in the way they operate. Finally I looked at the policies user's want, and compared it to their actual behavior when using the stores.

2.1. An SKB exporting statements to AppPAL.

App stores contain vast numbers of apps. To explore the properties and metadata surrounding the apps we needed a means to systematically analyse them, schedule the analysis tools to run and to store and retrieve results. This would help us organise future work looking at the apps in the stores, and provide a structure to do future research with.

As part of the AppPAL framework I imagined using static analysis results as part of constraints within the language. The SKB was designed to be lightweight and extendible allowing new tools to be added quickly. Implemented in Ruby the SKB supports running metadata-fetching (information scraped from inside the app or the Play store) and static-analysis tools in parallel over large collection of apps. Adding tools is quick and painless: add a library to Ruby, or a file to a configuration folder implementing a class interface—see Figure 4.

The SKB exports statements to AppPAL, and holds around 50,000 apps. Running the `skb dump` command causes the SKB to emit an AppPAL assertion context that can be used to make queries (Figure 3). The SKB can also emit statements from other principals where appropriate. For example in Figure 3 the SKB emitted statements from the Play store for the apps categorisation and review score. Since the information was obtained from the Play store the SKB issues statements as if the Play store had said them.

Development of the SKB is ongoing work. One possible extension of the SKB could be to store knowledge of vulnerabilities. A device could warn a user if they attempt to

```
$ skb dump
...
'skb' says 'apk://com.olbg.tipsandroid' can-act-as '34458db1a8d0d1e0bca9bb658cf79872867d97b8' .
'PlayStore' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasReviewScore('4.3').
'PlayStore' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCategory('Sports').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasPermission('android.permission.INTERNET').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasPermission('android.permission.ACCESS_NETWORK_STATE').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasPermission('android.permission.ACCESS_COARSE_LOCATION').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasPermission('android.permission.ACCESS_FINE_LOCATION').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasPermission('android.permission.WRITE_EXTERNAL_STORAGE').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasPermission('android.permission.READ_EXTERNAL_STORAGE').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateVersion('3 (0x2)').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateSerial_Number('2063699619 (0x7b018ea3)').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateSignature_Algorithm('sha256WithRSAEncryption').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateIssuer('O=Olbg').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateNot_Before('Aug 13 15:23:16 2013 GMT').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateNot_After('Aug  7 15:23:16 2038 GMT').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateSubject('O=Olbg').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificatePublic_Key_Algorithm('rsaEncryption').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificatePublic-Key('(2048 bit)').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateExponent('65537 (0x10001)').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateX509v3_Subject_Key_Identifier('').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateSignature_Algorithm('sha256WithRSAEncryption').
...
```

Figure 3: AppPAL output from the SKB.

install an app with a highly dangerous vulnerability, by checking that no bugs with a high CVSS score are known.

```
'user' says App requiresWarning
  if App hasCVSSScore(X)
  where X > 8.0.
```

The SKB is designed to make fetching and storing these kinds of statement easy. Extending the SKB with additional static analysis tools, and metadata fetching utilities is on-going work. We might also extend the SKB so queries can be made to it through AppPAL constraints. This might allow us to speed up checking some constraints (for example one involving running a static analysis tool) by retrieving the previously computed result from a server instead of recomputing. This would allow us to check policies faster (in some cases), which is desirable as long checking times may make AppPAL unusable.

2.2. AppPAL on Android

With a means to collect and store information about apps I wanted to be able to use the information collected to make policy decisions. In the first year I implemented a prototype AppPAL interpreter in Haskell. This was usable to start investigations but could not run on an Android device. Haskell lacks good ARM compilers, and cannot access Android libraries and APIs. I re-implemented it in Java, with a custom evaluation algorithm, and made sure it was portable, running as a library in an app, or Java program. The current implementation contains roughly the same number of lines of code, but runs everywhere and is significantly easier to extend.

Performance of the library when searching an assertion context for proofs is reasonable. The search procedure is at its slowest when performing large repeated delegations. Two synthetic benchmarks were created to check that the search procedure performed

```
require 'skb'

module ResultFetcher
  ##
  # Runs mallodroid on an APK
  class Mallodroid_HEAD < Skb::ResultFetcher
    ##
    # Timeout after 5 minutes
    def timeout
      300
    end

    def execute
      @out.puts `mallodroid
                 -x
                 -f "#{@apk.path}"`
      return true
    end
  end
end
```

```
require 'skb'
require 'zip/zip'

module MetaFetcher
  ##
  # Extracts the certificate used in the APK
  class Certificate < Skb::MetaFetcher
    def execute
      begin
        Zip::ZipFile.open @apk.path do |apk|
          cert = apk.find_entry \
            'META-INF/CERT.RSA'
          unless cert.nil?
            Dir.mktmpdir do |dir|
              cert.extract "#{dir}/CERT.RSA"
              out = `openssl pkcs7
                       -in '#{dir}/CERT.RSA'
                       -inform DERM
                       -print_certs |
                     openssl x509 -noout -text`
              @out << out
            end
            return true
          end
        end
        return false
      rescue => _e
        return false
      end
    end
  end
end
```

Figure 4: Fetching plugins for Mallodroid and the certificate information for the SKB.

Policy	Principals	Parse (s)	Check (s)
Straight	10	0.01	0.04
Straight	100	0.05	1.00
Straight	500	0.32	21.07
Straight	1000	0.43	87.75
Forking	256	0.10	1.93
Forking	512	0.23	7.47
Forking	1024	0.45	28.29
Forking	2048	0.85	117.68

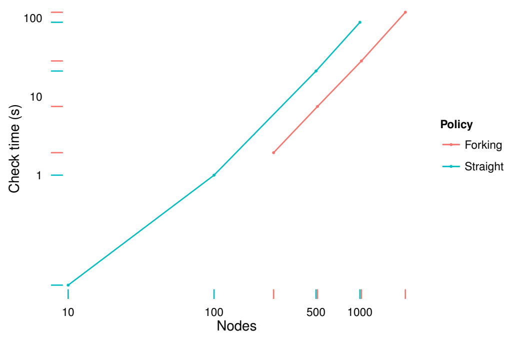


Figure 5: Bench-marking results on a Nexus 4 Android phone.

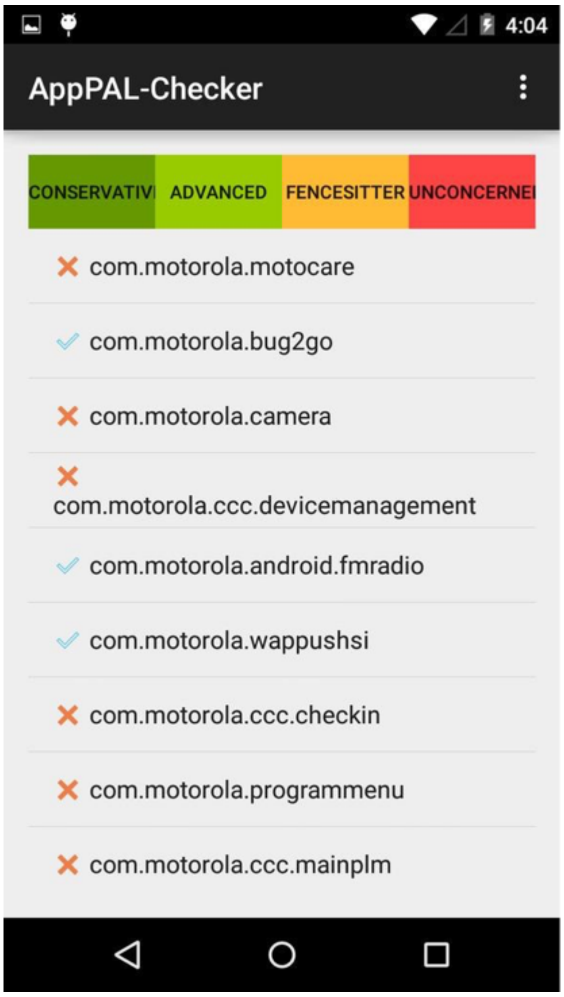
acceptably. Each benchmark consisted of repeated chain of delegations. The *straight* benchmark consists of a single long chain of delegation. The *forking* benchmark consists of a binary tree of principals delegating to each other. These benchmarks are reasonable as they model the worst kinds of policies to evaluate—though worse ones could be designed by forking even more.

On a *Nexus 4* phone checking times are measured in seconds when there are hundreds of delegations, and in minutes when there are thousands (Figure 5). We have only used a few delegations per decision when describing hypothetical user policies. Since the benchmarks show that long chains of delegation can be used, I believe the performance is acceptable. AppPAL constraints will be the slowest part of checking a policy: since a constraint can execute arbitrary programs or include network requests, the time it takes to run is independent of AppPAL's checking mechanisms. In practice the only ones I have used are permissions checks. These are very fast as they can call out to the Android package manager, or inspect the APK file directly. Constraints that take a long time to run (such as a static analysis tool) could be used but this may lead to policies designed to limit the number of constraint checks, as they may take too long to check. Current practical policies check small numbers of permissions per app. Since the numbers of constraint checks used in policies are small it is not worth optimising AppPAL's evaluation algorithm to handle checking large numbers of slow to run constraints (yet).

The library has been integrated a proof-of-concept policy checking app that scans installed apps against four user privacy policies (Figure 6). This may be extended into something more comprehensive, or into a more polished demo to help promote my research.

2.3. Explicit policies in stores

With a means of enforcing policies I wanted to look at the policies already being enforced by the marketplaces. One source of these policies is in the user and developer agreements required to use or sell apps in the stores. These policies should give us a good idea about how the stores are used as they are explicit contracts between users, developers and the



✗ com.motorola.motocare
✓ com.motorola.bug2go
✗ com.motorola.camera
✗ com.motorola.ccc.devicemanagement
✓ com.motorola.android.fmradio
✓ com.motorola.wappushsi
✗ com.motorola.ccc.checkin
✗ com.motorola.programmenu
✗ com.motorola.ccc.mainplm

Figure 6: Screen shot from AppPAL checking app.

stores themselves. We might also be able to describe the differences between stores using AppPAL, to help users pick a store that meets their own policies.

There are many Android app stores available. I believed that each of the different stores would have different policies for users and developers, and with different privacy conditions. I read the privacy, developer and user policies for four different app stores (Google Play, Amazon, Aptoide and Yandex) to compare them. In practice I found that the policies tended to be very similar; perhaps even copied in some places. There were some differences when it came to payment processing, and minimum ages to use the store.

Some stores (excepting Google's) kept some right to modify any apps, typically for advertising. This might form the basis of interesting trust relationships. To install an app Android requires its code is signed by a developer. Modifying an app would mean the signature is invalid and so the app would need to be re-signed by the store to be installed. Re-signing is interesting because the trust in the authenticity of the app moves from the developer to the store.

A summary of the different terms and conditions is in Figure 7.

This work could be extended to help users pick an app store that is right for them using a policy. Projects like COAT [5] aim to help users pick a store based on their privacy requirements. We could imagine a similar scheme where AppPAL helps developers decide which stores to submit their apps to, and which stores users should buy from. For example a developer may only agree to sell their app on a store if the store lets them set their own price and they get a 75% cut of the profits.

```
'dev' says 'selling-policy' isMetBy(Store)
  if 'dev' setsPriceIn(Store),
    Cut isCutIn(Store).
  where Cut > 0.75.
```

Figure 7 could be translated into AppPAL and queried to decide which stores can sell the apps; and with this policy only Aptoide would be acceptable.

In Europe and America the choice of which market to use is less interesting. Only Google's Play store (and to a lesser extent Amazon's app store) has enough market share to be compelling. In China, however, where the Play store is banned by the government, there is a greater choice of marketplace and this becomes a more compelling avenue of research.

2.4. Distribution mechanisms in app stores.

Whilst the user and developer agreements describe what the apps stores say they will do; to see what do in practice we need reverse engineering. A user buying an app probably wants it sent to their device. Using an SSL proxy I started looking at the distribution mechanisms, and started reverse engineering their protocols. For simple website stores, such as Opera's mobile store, where apps are distributed without encryption it is trivial to modify downloaded apps on the fly. A man in the middle (MITM) attack using a tool like can `mitmproxy` can modify the apps as they are downloaded. In these stores there was no verification that the app downloaded matched the app requested.

	Google Play	Amazon	Yandex	Aptoide
User ID	Name address and billing details.	Amazon ID.	None for free apps, payment details for paid.	Contact details. No verification but agreement not to lie.
Client info taken	Installation data, device ID, browsing history, cookies. Can opt out.	Device ID, network info, location, usage data.	Device ID, SIM number, Device content, System data, browsing history.	Transaction history. They may share it with developers.
Customer Payments	Google Wallet and others at Google's discretion.	Amazon.	Approved processor by Yandex or store operator.	Approved processor by Aptoide.
Who is paid?	Google Commerce.	Amazon.	Developer.	Store owner.
Prices set by	Developer.	Amazon.	Developer (but Yandex may restrict to set values).	Developer and store owner.
Refunds	Only for defective or removed content. A refund may be requested for two hours after purchase.	No.	Up to 15 minutes after purchase. No for IAP.	Up to 24 hours after purchase.
Age of use	At least 13.	Any age (with consent of guardian). No alcohol related content below 21.	At least 14.	A legal age to form a contract with Aptoide.
Update provision	You agree to receive updates.	By default.	Yes for security and bug-fixes.	Yes agree to receive updates.
Moderation	No obligation (but they may).	Publisher obliged to provide info which may be used to give ratings. Amazon will not check these ratings are accurate.	No obligation (but they may).	No obligation (but they may). Trusted app mark does not indicate moderation.
Acceptable use	No use as part of a public performance, or for dangerous activities where failure may lead to death.			No modification, rental, distribution or derivative works. You may use the software.
Store rights to app	Marketing and optimising Android.	Distribution, evaluation, modification, advertising, and creating derivatives for promotion.	Advertising.	Modification and re-selling.
Withdrawal from sale	Immediate.	10 days, or 5 days if for copy-write reasons.	90 days. A copy will be retained.	You may.
Developer ID	Google account and billing details.	Amazon ID.	Email, company name, tax-id.	Email (preferably a Google developer one).
EULA	Default offered.	Only if it doesn't interfere with Amazon's terms.	Must be provided.	Default offered
Content restrictions	No alternate stores, sexual, violence, IP infringing, PII publishing, illegal, gambling, malware, self-modifying or system modifying content. No unpredictable network use.	No offensive, pornography, illegal, IP infringing or gambling content.	No defects. No illegal, disruptive, sexual, IP infringing, PII stealing, alternative stores, or open-source content.	No displaying or linking to illegal, privacy interfering, violent, PII stealing, IP infringing content. Nothing *spammy* or with unpredictable network use.
Payout rates	70% of the user's payment.	70% list price (minus card fees).	70% net-revenue (minus card fees).	75% revenue share (minus card fees).

Figure 7: Summary of conditions in different stores.

Some issues were found in other stores. Amazon's store was the only one to enable certificate pinning. Certificate pinning can prevent SSL sniffing attacks as it requires that an SSL certificate is one known to the app; rather than one that exists in the device's root of trust. The only pinned certificate, however, was for accessing the login server. Once the login token had been issued, however, the traffic between the store and the device could be intercepted and decrypted freely. This isn't a vulnerability as there were other mechanisms in place within the store to prevent app modification: namely cryptographics signatures that came with every app. It does hint, however, that the login system and the store itself have different security guarantees, which may lead to other problems within the store.

Google's store seemed to sometimes drop encryption when downloading APK files. To download an app from the store the user has to follow a protocol, which we describe abstractly in Figure 8. This was inferred by looking at the network traffic and reverse-engineering the store's APK file. After completing the purchase the user presents their proof-of-purchase token to the store along with a description of the app and their own ID (messages 5 & 6) The store then tells them a server to speak to which provides the app to them without further information (the blank message 7 and message 8). This is implemented using a URL redirect, where S and S' represent the servers the client talks to. What we observed, however, is that occasionally the redirect was to a server using HTTP rather than HTTPS. This meant that an attacker sniffing the traffic could see which apps were being downloaded and take a copy for themselves. If the sniffing party was an employer, and usage of the app gave away personal information (say a diabetes app or Grindr), then this could be problematic. By replaying message seven the user (or a third-party who knew the URL) could re-download the app for up to a week later.

Each of the stores I looked at had a slightly different protocol for buying apps and a different means of downloading them onto the device. In an AppPAL-enhanced app store we might imagine apps being supplied with some assertions about their behaviour.

One way to implement this would be for the store, as part of the download, give the user a bundle of AppPAL statements. The user could filter these to find the statements that are useful for deciding any policies they may have. This force the mobile device to do extra processing which is undesirable if running on batteries. Working out how to send the AppPAL statements in the download protocols, (and with knowledge distribution in general, proposed in Subsection 3.1) seems interesting.

2.5. User app installation behaviour

Having explored the policies and protocols in the stores we also wanted to explore how the stores are used in practice. Users have policies about what data apps should be collecting. Lin et al. [10] identified four different user-privacy policies; however they did not look at whether they enforced them in practice. Using AppPAL I implemented a simplification of the policies Lin et al. identified. The Carat project [11] collected app installation data from users who installed their energy tracking app. They shared this data with us in an anonymised form where I could see who had installed what; provided I knew the package names of the apps they installed. Using the SKB I de-anonymised

1.	$C \longrightarrow S$:	U, C, a_{d}
2.	$S \longrightarrow C$:	$a_{\mathrm{d}}, ?$
3.	$C \longrightarrow S$:	$U, !$
4.	$S \longrightarrow C$:	$a_{\mathrm{d}}, \$$
5.	$C \longrightarrow S$:	$U, a_{\mathrm{d}}, \$$
6.	$S \longrightarrow C$:	S'
7.	$C \longrightarrow S'$:	
8.	$S' \longrightarrow C$:	a

In message 6, S redirects C using a 302-redirect message. By replaying message 7 I found I could re-download the app on a different client. I found that the redirection to the other server sometimes did not use SSL.

Symbol	Meaning
S	Store (`android.clients.google.com`).
C	Client the app store app running on the phone.
U	User, identified by a token.
a	An app.
a_{d}	a description of the app a.
$?$	Purchase challenge
$!$	Purchase challenge response (seemingly derived from $?$ and U).
$\$$	Download token.
S'	Alternate store URL

Figure 8: Abstraction of the protocol used by Google's Play store to purchase an app.

4,300 apps (5%) which accounted for 50% of all app installs. I selected 44,000 users for whom I knew 20 or more app installs.

With the Carat data I measured the extent each user conformed to the privacy policies. I also took a list of known malware and potentially unwanted programs (PUPS) from McAfee to measure the extent of malware installations on Android. I tested whether users who enforce their privacy policies install less malware.

I found that very few users follow a privacy policy all the time [8]. A few users, however, seem to be installing apps meeting a policy most of the time. For the unconcerned policy (the most permissive) only 1,606 users (4%) had 50% compliance; and only 120 users (0.3%) had 80% compliance. For the stricter conservative policy only 60 users were complying half the time, and just 7 more than 80% of the time. This suggests that while users may have privacy preferences the majority are not attempting to enforce them.

I found 1% of the users had a PUP or malicious app installed. A user is three times more likely to have a PUP installed than malware. Only nine users had both a PUP and malware installed. Users who were complying more than half the time with the conservative or advanced policies complied with the malware or PUP policies fully. This is significant (P-value < 0.05) and suggests that users who pick their apps carefully are less likely to experience malware.

This work was presented as a poster at SOUPS [8], and I am aiming to publish it as part of a paper targeting the ESSoS conference (a draft of which is attached in an appendix).

3. Suggested topics for next year

In the thesis proposal I suggested that in the third year I might look at advanced kinds of policies. I also wanted to look at what might happen when policies have to deal with updates, to policies and apps on the device. Some topics included:

- The app collusion problem.
- What happens when policies are composed?
- Policy revocation and modification.
- App update policies.

Some of these problems have become less interesting: for example app updates. Various different policies could be encoded in AppPAL for the precise behaviour of an update. Since most stores mandate accepting updates in the user agreement, and the default behaviour is to auto-install updates, these schemes are unrealistic. Updates typically include bug-fixes so it is in the user's interests to accept them (despite what they may believe [15]). Implementing methods to weaken security does not seem to be worthwhile. Using AppPAL as a modelling language to precisely describe the trust mechanisms and delegations in an update is interesting. It might make sense to include a comparison of the different update mechanisms in Android and iOS in the thesis.

Policy modification is less interesting. Apps should be rechecked before they can be allowed to run again. If they now break the policy, the user should be prompted to remove the app or make an exception. Results from long-running analysis tools can be stored by the SKB. These could be reused by AppPALs constraint mechanisms to avoid re-computation, if no other changes were made.

The app collusion problem is interesting. Whilst it is conceivable that AppPAL policies could describe and prevent attacks, to actually implement the protection in a way that could be used would probably require modifications to the intent system, and potentially binder (Android's primary IPC mechanism). This probably makes it out of scope of the PhD as we are looking at Android, not a modification of it.

The mechanisms for mechanically composing policies in AppPAL are trivial[3] but what might happen when two principals actively disagree is more interesting. AppPAL doesn't have support for negation, but it feels right that there should be some way to distinguish between a principal saying an app meets, does not meet, or does not know whether it meets a policy. Work next year should look at the protocols for distributing AppPAL statements.

3.1. Knowledge distribution protocol

AppPAL can enforce policies and I have started looking at the kinds of policies users might want to use. When describing AppPAL policies I have thought about delegation

[3] `'user'` *says* `'composed-policy' isMetBy(App)`*if* `'pol1' isMetBy(App), 'pol2' isMetBy(App).`

relationships and how different principals can be trusted to help make decisions. What isn't necessarily clear is how these principals get to make these statements and how they are transferred from speaker to speaker.

On a memory constrained device, like a mobile phone, storing a database of all possible AppPAL statements by every speaker is not viable. There are almost 1.5 million apps available on the Google Play store not including other stores or multiple versions of the same app. Storing data on all apps will not work as the file size will be huge. In the papers on SecPAL [3, 2] (on which AppPAL is based) there is little talk about how the knowledge should be distributed. **(author?)** describe how principals should sign their AppPAL statements to identify themselves as having said them, and how an X.509-style public key system could be used to tie keys to principals. This tells us how to check the statements are not forged but does not give the distribution mechanisms. Related languages [4, 1, 6, 7] extended SecPAL features but still didn't specify the distribution protocol. SecPAL was built to be a distributed language with principals delegating statements. Without a distribution protocols some scenarios can become difficult.

Consider Alice who wants to install an app on her phone. Her installation policy requires confirmation the app is not malware. She knows that McAfee can say (possibly with further delegation) whether an app is malicious or not; but this is a new app and she does not have any prior information about it. She needs to get more information.

One, very simple, approach to distribute the knowledge might be to use a protocol such as:

1.	$U \longrightarrow S$:	$q, d?, m$
2.	$S \longrightarrow U$:	$r, t, \{r, t\}_{S_{SK}}$

A user (U) wishes to find out if a server (S) knows any information that will help them answer a query (q). To do this they send the server the query along with any metadata they have, for example the app itself, to the server and say whether they'll allow the server to delegate ($d?$). In reply the server sends any information it thinks would be helpful (r), plus a timestamp (t) and a signature or the response and timestamp so the user is sure they're recieving fresh information. If the user finds they need more information then the protocol will need to be replayed.

This gives a simple protocol for distributing knowledge but it doesn't deal with all the complexities of knowledge distribution. In particular:

- If the app is new McAfee may not know about it either. How should she send it to them for analysis? In the protocol we suggested the app itself could be sent as part of the metadata relating to the query. There may be legal issues surrounding Alice redistributing the app to McAfee, however. If the app is large and Alice has a limited data-allowance she may not want to send the whole app if it is large. She could just send a link to the app on an app store, but McAfee may not want to check it if it costs money to download, or the terms of buying the app prevent reverse engineering.

- How do McAfee respond when they can't answer the query? AppPAL does not contain negation but in this scenario it would be helpful to distinguish a statement from McAfee that the app is safe, from one where they know it is definitely malware, from one where they are unsure and wish to err on the side of caution and not make a definite statement.

- What should Alice do while she waits: keep waiting, fail and keep a note to recheck later or fail and never ask again about the app? Should she say how long she is willing to wait so McAfee can distinguish between needing an answer now (Alice is trying to use the app right now) or that the answer isnt urgent (her phone is caching information about apps she hasn't bought yet incase she buys one of them in future).

- If McAfee wish to delegate the decision they could issue another *can-say* statement. Should McAfee send the public key and any statements from the delegated party to the user, or just the server address and a reference to the public key on a PKI server?

- When should Alice ask McAfee? Before any evaluation would be the easiest time as it would require no change to the evaluation algorithm but during the evaluation might be more appropriate as statements could be imported as needed.

Work this year should look at developing a strategy, and defining a protocol to share and acquire knowledge through AppPAL statements on demand. This would be a worthwhile contribution, as it is novel, and would help extend AppPAL from a SecPAL instantiation to its own language. To do this we aim to complete the following goals:

- Define a protocol where a principal can query another principal for information. The protocol must define precisely what should happen when a further delegation occurs.

- Show that AppPAL (or SecPAL) is still sound and complete when using this protocol as part of evaluation.

- Integrate the protocol with the AppPAL library and check the performance impact from using it. Explore different strategies for what to do while waiting for answers, and how AppPAL could be configured to switch between them.

- Demonstrate use of the protocol using the SKB as a third-party who can be queried for information.

3.2. Case study

Having an authorization logic model and enforce a larger, currently implemented policy shows the limitations and expressivity of a language. It also shows that it is applicable to current compliance problems and is more than just a toy. AppPAL policies I have used so far have been synthetic. This has been fine for testing and demonstrating the language, but a larger example drawn from a real security policy would help give AppPAL credibility.

A natural source for this policy is a corporate BYOD policy where employers restrict the usage of mobile devices in the workplace. The US National Institute of Standards and Technology (NIST) have published recommendations for mobile device security and BYOD in the workplace [14, 13]. Translating the recommendations into AppPAL and showing their enforcement would make a good case study. The study would focus on any difficulties in translating the policies and the extent the recommendations could be enforced automatically. Specifically we would aim to:

- Present an AppPAL policy derived from the guidelines in Sections 2–3 of NIST-SP-800-46 (which gives guidelines for telework and remote access mobile security) and Section 2 of NIST-SP-800-124 (guidelines for mobile security).
- Show where the strengths and limits of AppPAL when enforcing the guidelines.
- It is reasonable to expect that we cannot automatically enforce every aspect of the guidelines as they are high level and written in natural language. Some aspects of the guidelines will need a human to certify that it is met. Using AppPAL we can show where a human will be needed and measure the extend of automatic enforcement.

4. Plan for thesis submission

Hypothesis. Automatic tools and policy languages would provide a better means of enforcement for mobile device policies than existing mechanisms which rely on manual inspection.

I have 18 months remaining, with my funding running out in March 2017. My plans for the time remaining are shown in Figure 9. The next year will be primarily spent completing the projects described in Section 3; and continuing any other work that seems interesting as opportunities arise. Once my review is completed I plan to spend at least one day a week writing up work for my thesis; focussing on the literature review, and completed investigations. From October 2016 I plan to spend the majority of my time writing (as suggested in my thesis proposal). From January 2017 I plan to be predominantly focussing on editing. I believe this is a reasonable plan based on my writing speed, and the time I typically spend editing documents.

Below is a proposed table of contents for the thesis, with content suggestions for each chapter.

1. Introduction
2. Mobile ecosystems

   App stores and mobile app deployment

   - Introduce app stores and current development practices.

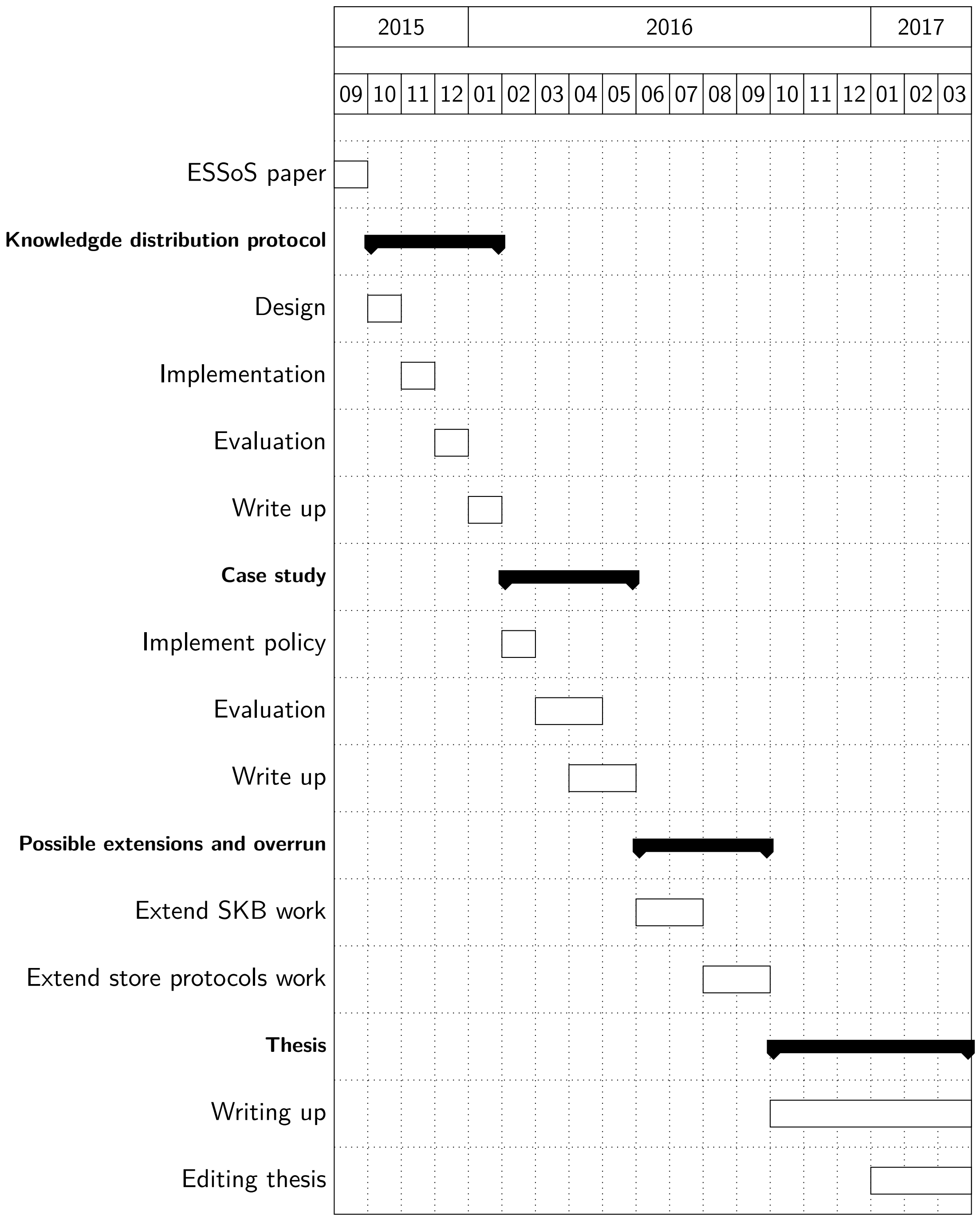


Figure 9: Gantt chart showing plan for time remaining.

    - Show the differences between different market places and the platforms (iOS vs Android).

   Android Security Model

    - Give overview of Android security architecture.
    - Describe how apps work and interact with the system from a technical perspective.
    - Introduce the android permissions model and how different apps can access functionality provided by the platform.
    - Show how app signing works.

   Android policies

    - Introduce the need for policies and the motivation for the PhD.
    - Show how users do not seem to be following their privacy policies.
    - Explain how some companies have mobile device policies, and how users are frustrated with data leaks.
    - Show existing tools, such as Kirin, and explain why they're not good enough.

3. AppPAL Implementation

   The need for a policy language

    - Introduce scenarios where AppPAL can be used to enforce a policy
    - Show trust relationships between different principals (a user at work).
    - Start to introduce the language.

   Design and implementation of AppPAL

    - Formally present the language, as an instantiation of SecPAL.
    - Show evaluation algorithm.

4. AppPAL Experimentation

   Deployment

    - Show applications using AppPAL.
    - Stores generated by policy.
    - On device policy checking.
    - Device configuration by policy (if I can get access to Android M features)

   Distribution

    - Define protocol for knowledge acquisition.
    - Describe implementation of protocol.
    - Implement the protocol as part of AppPAL's evaluation algorithm.

5. Evaluation

   User privacy policies

   - Show privacy paradox with Android apps.
   - Use Carat (and McAfee?) data as a model of what users have installed.
   - Show power of AppPAL as a query and modelling language.
   - Discuss the policies users do seem to be enforcing in practice.

   Case study

   - Present NIST guidelines for mobile device guidelines for managing the security of mobile devices in the enterprise expressed in AppPAL.
   - Show the language is expressive enough to describe enterprise policies.
   - Discuss problems translating the recommendations, and where the guidelines could not be checked without manual intervention.

6. Related work

7. Future work

References

[1] B Aziz, A Arenas, and M Wilson. SecPAL4DSA. *Cloud Computing and Intelligence Systems*, 2011.

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

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

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

[5] C Fernandez-Gago, V Tountopoulos, S Fischer-Hübner, R Alnemr, D Nuñez, J Angulo, T Pulls, and T Koulouris. Tools for Cloud Accountability. *IFIP Privacy and Identity*, 2015.

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

[7] Y Gurevich and I Neeman. DKAL 2. Technical Report MSR-TR-2009-11, Microsoft Research, February 2009.

[8] J Hallett and D Aspinall. Poster: Using Authorization Logic to Capture User Policies in Mobile Ecosystems. In *Symposium On Usable Privacy and Security*, pages 1–2, July 2015.

[9] P G Kelley, L F Cranor, and N Sadeh. Privacy as part of the app decision-making process. In *Special Interest Group on Computer-Human Interaction*, pages 3393–3402, New York, New York, USA, April 2013. ACM Request Permissions.

[10] J Lin, B Liu, N Sadeh, and J I Hong. Modeling Users' Mobile App Privacy Preferences. *Symposium On Usable Privacy and Security*, 2014.

[11] A J Oliner, A P Iyer, I Stoica, and E Lagerspetz. Carat: Collaborative energy diagnosis for mobile devices. In *Embedded Network Sensor Systems*, 2013.

[12] W Prata, A de Moraes, and M Quaresma. User's demography and expectation regarding search, purchase and evaluation in mobile application store - Work: A Journal of Prevention, Assessment and Rehabilitation - Volume 41, Supplement 1/2012 - IOS Press. *Work: A Journal of Prevention*, 2012.

[13] K Scarfone, P Hoffman, and M Souppaya. NIST Special Publication 800-46: Guide to Enterprise Telework and Remote Access Security, June 2009.

[14] M Souppaya and K Scarfone. NIST Special Publication 800-124: Guidelines for Managing the Security of Mobile Devices in the Enterprise, June 2013.

[15] K Vaniea, E Rader, and R Wash. Betrayed by updates: how negative experiences affect future security. In *Computer Human Interaction*, pages 2671–2674, New York, New York, USA, April 2014. ACM.

A. Paper for ESSoS 2016

The attached paper is a draft being prepared for the ESSoS[4] symposium. The paper will introduce the AppPAL language and its implementation. I give sanity checking benchmarks to show its performance is not unreasonable. Finally, I demonstrate AppPAL by presenting our work using AppPAL to filter user's privacy policies from our SOUPS poster, as part of a full paper.

[4]Engineering Secure Software and Systems