Authorization Logic for Mobile Ecosystems

Second Year Report

Joseph Hallett

August 26, 2015

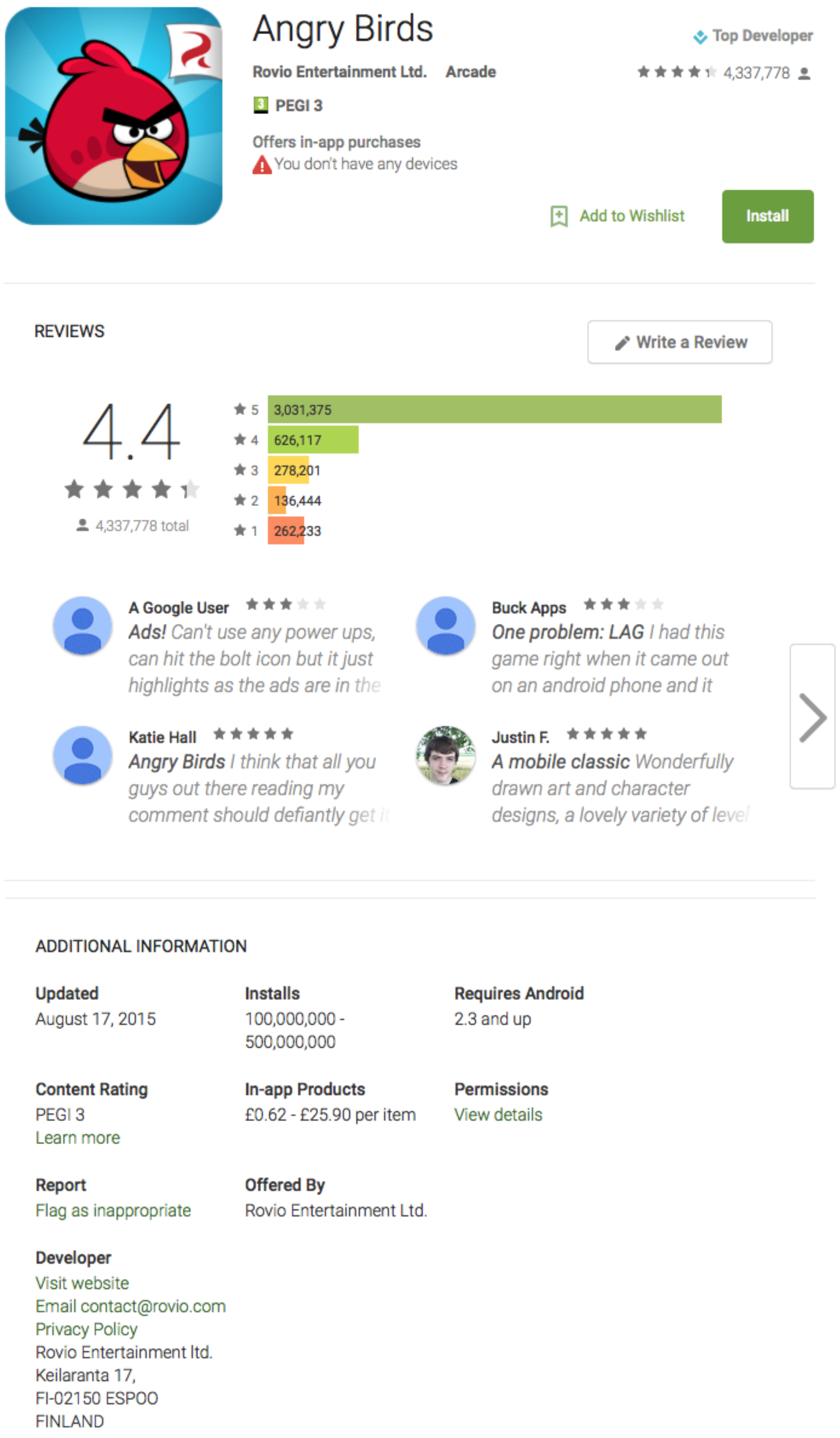
This report describes the second year work of my PhD. I give a brief overview of app installation policies and the AppPAL authorisation logic; before describing the work completed this year. I review topics suggested in the thesis proposal; and describe my work re-implementing AppPAL, exploring store 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 for the proposed thesis.

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 predominantly only run software sold through stores the manufacturer control. People take them to work and on holiday; into their homes and out into town or the countryside. These devices communicate through wireless technologies 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 these devices to operate in.

The interaction between stores, device users, apps and phones create trust relationships between each of the entities. Users trust the stores to sell them 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 it, 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



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	**Installs**	**Requires Android**
August 17, 2015	100,000,000 - 500,000,000	2.3 and up
Content Rating	**In-app Products**	**Permissions**
PEGI 3 Learn more	£0.62 - £25.90 per item	View details
Report	**Offered By**	
Flag as inappropriate	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.

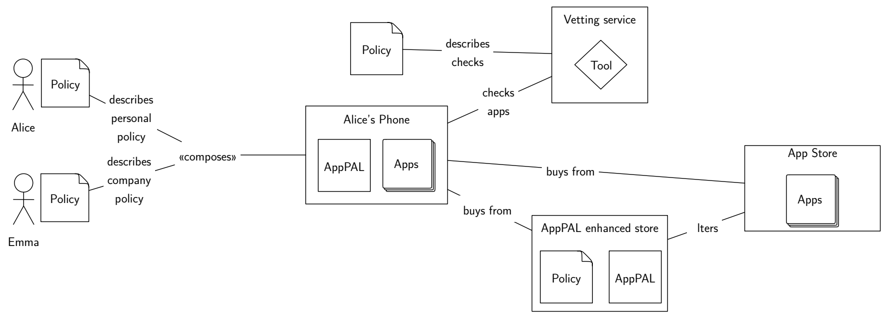


Figure 2: The mobile ecosystem surrounding a phone used for BYOD and the trust relationships surrounding it.

want and to find the apps which suit how they want to use their device. This can be difficult as it isn't obvious how apps use the data each has access to. App stores offer a wide range of software for users to choose. Users pick particular apps for a variety of reasons: ordering in the store [12], reviews and privacy concerns [9], and security rules from their employer. These policies could be written down and used to guide device owners when they are making a decision that goes against them.

If an employer uses a bring your own device (BYOD) scheme at 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*[1] the employer can restrict the use of specific apps. This is a very unsubtle form of usage control; it doesn't allow for rich policies using information from third parties or on the basis of metadata about the app; or 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 all they have to enforce their policies. For more advanced policies, such as not leaking company documents, compliance is often left to employees.

Policies exist for mobile ecosystems; but there aren't ways of modelling them precisely or enforcing them. I believe that authorization logic can be used to describe these policies and enforce them automatically. Authorization logic's can describe the trust relationships and policies surrounding Android formally. This lets us to make precise statements about the security of these systems. This 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 explicitly as policies they can be enforced automatically. By checking the policies we can enforce them at run-time; or warn users when making decisions that go against their policies. 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

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

decisions in mobile devices. 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 SecPAL [3] we have developed to model and enforce app installation policies. Using AppPAL we can describe policies with trust relationships, and which incorporate constraints. This gives us a powerful language which can be used to describe user's and companies app policies precisely.

AppPAL can express that a user finds they can install an AppPAL:

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

Or that an app is installable *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.
```

Decisions can be delegated to other users; with separate versions where further delegation is allowed (*inf*) or not (*0*).

```
'user' says 'it-department' can-say 0 'office-app-policy' isMetBy(App).
```

And principals can be given roles or subjects renamed.

```
'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].

2. Summary of second year work

I proposed primarily looking at the role of app stores this year in my thesis proposal. Specifically I wanted to understand how users interacted with the stores. This would enable us to better understand user behaviour. By better understanding user behaviour I could write policies that accurately described the users. 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.

Several of these topics changed as year went on. I implemented a system for creating app stores based on an AppPAL policy, but when I also gained access to real installation data. Using this data I started to look at what apps users pick from stores and the kinds of policies they apply. 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 one and have used the data collected from it in other aspects of the project.

In the next few sections I describe work I have completed in the second year; and show findings.

2.1. Re-implemented AppPAL in Java.

In the first year we implemented a prototype AppPAL interpreter in Haskell. Haskell lacks good ARM compilers, and cannot access Android libraries and APIs easily. This made on device experimentation harder. I re-implemented it in Java and made sure it was portable, running 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 modify.

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 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 device checking times are measured in seconds when there are hundreds of delegations in a single policy check, and in minutes when there are thousands (Figure 3). 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 their performance is independent of AppPAL. In practice the only ones I have used (so far) are permissions checks. These are very fast as they can call out to the Android package manager. Slowly evaluating constraints could be used but this may lead to policy designed to limit the number of constraint checks. Current practical policies check five or six permissions per app. Since the numbers of checks are small it is not worth optimising (yet).

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

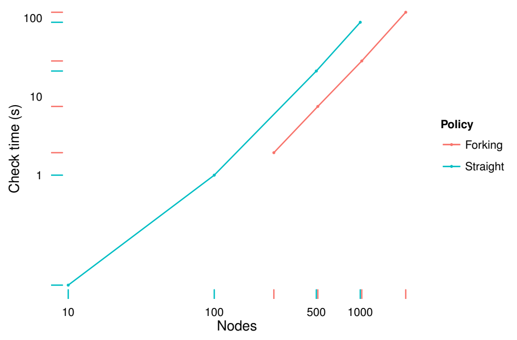


Figure 3: Bench-marking results on a Nexus 4 Android phone.

2.2. Built a security knowledge base (SKB) exporting statements to AppPAL.

As part of the AppPAL framework I imagined using results from static analysis tools as constraints within the language. To collect and store these results I built an extensible SKB that was lightweight and which could be trivially extended. Implemented in Ruby the SKB supports running metadata-fetching 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 5.

The SKB exports statements to AppPAL, and holds around 40,000 apps at the moment. Running the `skb dump` command causes the SKB to emit an AppPAL assertion context that can be used to make queries (Figure 4). The SKB can also emit statements from other principals where appropriate. For example in Figure 4 the SKB emitted statements from the Play store for the apps categorisation and review score. Since the information was obtained from the Play store it makes sense to speak as it in this scenario.

Development of the SKB is ongoing work. One possible extension of the SKB could be to store knowledge of vulnerabilities. Hypothetically a device could warn a user if they attempt to install an app with a highly dangerous vulnerability.

```
'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. Its development will continue, as needed, into the next year.

2.3. Explored existing app store privacy policies.

There are many Android app stores available. I believed that each of the different stores would have different policies for kinds of apps, developers, and 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

```
$ 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 4: AppPAL output from the SKB.

```ruby
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
```

```ruby
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 5: Fetching plugins for Mallodroid and the certificate information for the SKB.

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 as 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 6.

This work could be extended to help users pick an app store that is right for them, on the basis of 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 is used to help 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 6 could be translated into AppPAL to decide which stores can sell the apps; and on the basis of this policy only Aptoide would be acceptable.

In Europe and America the choice of which market to use is not interesting as only Google's (and to an extent Amazon's) has enough market share to be compelling, and is pre-installed on most devices. 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. Explored distribution mechanisms in existing stores.

A user buying an app probably wants the app sent to their device. Using an SSL proxy I started looking at the precise 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.

Various security issues were found in other stores. Amazon's store was the only one to enable certificate pinning. The only pinned certificate was for accessing the login server. Once the user had logged on, an SSL proxy could be used to examine traffic. This meant that the store prevented you from logging on to the store when traffic was modified in a MITM attack. 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 in itself bad as certificate pinning is not often required (though it probably should be in this scenario) but is interesting.

Google's store occasionally seemed to drop encryption when downloading APK files. To download an app from the store the user has to follow a protocol, which we describe

	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 bellow 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 user's payment.	70% list price (minus card fees).	70% net-revenue (minus card fees).	75% revenue share (minus card fees).

Figure 6: Summary of conditions in different stores.

in abstract terms in Figure 7. This was inferred by looking at the network traffic and reverse-engineering the store APK file. After completing the purchase the user presents their proof-of-purchase token to the store along with a description of the app their identifier (messages 5 & 6) The store tells 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 in practice using a URL redirect, where S and S' are servers. 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 could be problematic. By replaying message seven the user (or a third-party who sniffed the URL) could re-download the app for up to a week later.

I would like to continue this work into the next year. 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. I might imagine in an AppPAL-enhanced app store apps being supplied with some assertions about their behaviour. Working out how to include these in the download protocols, (and with knowledge distribution in general, proposed in Subsection 3.1) seems interesting.

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

2.5. User app installation behaviour

Mobile device users have policies about what data apps should be able to collected. Lin et al. [10] identified four different privacy policies when users think about apps; 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. Ignoring system apps, using the SKB I de-anonymised 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.

Using the Carat data I measured the extent each user conformed with 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. From this I tested whether users who enforce their privacy policies install less malware.

I found that very few users follow these policies all the time. A few users, however, do seem to be installing apps meeting a policies most of the time (Figure 8a). 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 (Figure 8b). 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 (Figure 8c). This is significant (P-value < 0.05) and suggests that users who pick their apps carefully are less likely to experience malware.

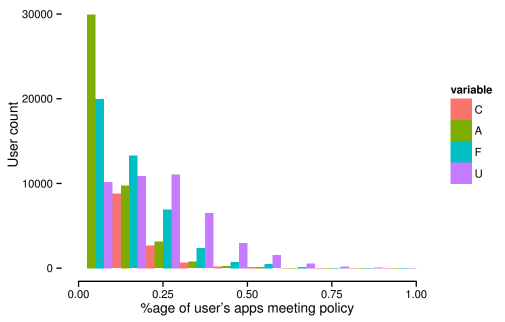
This work was presented as a poster at SOUPS [8], and I are aiming to publish it fully as part of a paper targeting the ESSoS conference (a draft of which is attached in an appendix).

3. Next directions

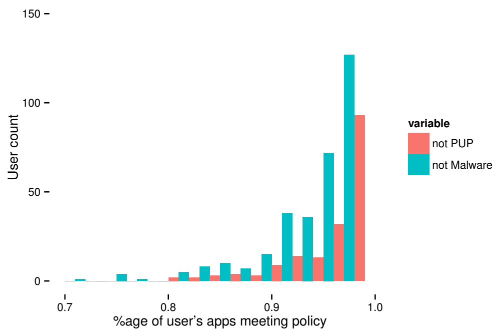
In the thesis proposal I suggested that in the third year I might look at advanced kinds of policies. I also wanted to look at what might happen when policies have to deal with updates, to policies and apps on the device. Some included:

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

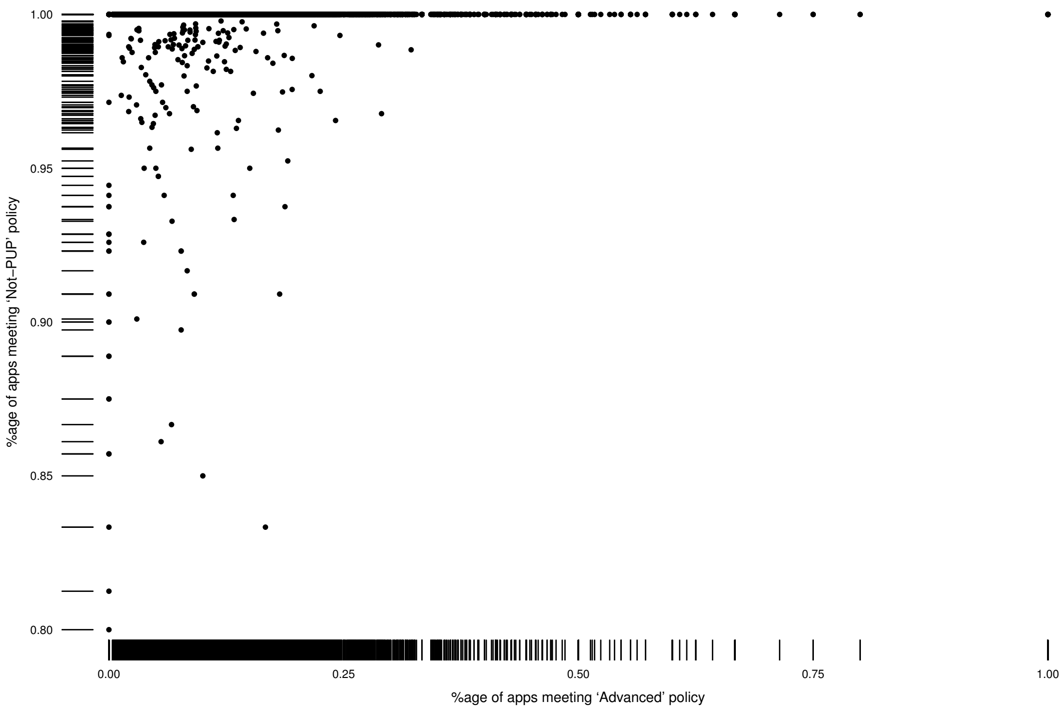
Some of these problems have become less interesting: for example app updates. Various different policies for updates could be encoded in AppPAL for the precise behaviour of an update. Since most stores mandate accepting updates in the user agreement and the default behaviour is to auto-install updates schemes are unrealistic. Updates typically



(a) User conformance to identified privacy policies.



(b) User malware installation rates.



(c) Comparison of users following the advanced policy with malware installation.

Figure 8: Results from study on user app installation behaviour.

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 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 static 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) and this probably makes it out of scope of the PhD.

The mechanisms for mechanically composing policies in AppPAL are trivial[2] 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. Continuing research should continue along with 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 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 ground 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. In the papers on SecPAL [3, 2] (on which AppPAL is based) there is little talk about how the knowledge should be distributed. They 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 doesn't give the distribution mechanisms. Related languages [4, 1, 6, 7] extended SecPAL language features but also 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

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

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. This scenario raises more questions:

- How does she ask McAfee about the app? What is the protocol for speaking and distributing statements?
- If the app is new McAfee may not know about it either. How should she send it to them for analysis? What should she do while she waits: keep waiting, fail and keep a note to recheck later or fail and never ask again about the app? Are there legal issues surrounding Alice redistributing the app to McAfee?
- How do McAfee respond? 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.
- 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.

3.2. Case study

Having an authorization logic model and enforce a real world policy shows the limitations and expressivity of a language. It also shows that it is applicable to current compliance problems and solves *real world problems* rather than just the policies I have learnt from user behaviour.

A natural source of these real world policies is corporate BYOD policies, where employers describe restrictions on 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 mobile device relevant sections into AppPAL and showing their enforcement might make a good case study. The study would focus on any difficulties in translating the policies and the extent the policy could be enforced automatically.

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 policy would help give AppPAL credibility.

4. Proposed thesis outline

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.

1. Introduction
2. Mobile ecosystems

   App stores and mobile app deployment

   - Introduce app stores and current development practices.
   - Show the differences between different market places and the platforms (iOS vs Android).

   Android Security Model

   - 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.

5. Evaluation

User privacy policies

- Show privacy paradox with Android apps.
- Use Carat (and McAfee?) data as a model of what users have installed.
- Shows power of AppPAL as a query and modelling language.

Case study

- Show how I can take a corporate policy and enforce it using AppPAL.
- Show the language is expressive enough to describe real policy scenarios.
- Shows that AppPAL can describe corporate policies that someone might want to use rather than synthetic examples.

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[3] 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.

[3]Engineering Secure Software and Systems