Authorization Logic for App Policies

Second Year Report

Joseph Hallett

August 21, 2015

This document describes progress made in the second year of the PhD studies. We give a brief overview of the topic before describing the work completed this year, and how it differs from the first year report. We briefly explain work reimplementing AppPAL, exploring store and user policies and looking at protocols for app distribution. We say what we expect to explore in the final year. Finally we give a table of contents for the proposed thesis.

1 Introduction

Mobile devices let users pick the apps they want to run. 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 [7], reviews and privacy concerns [4], and security rules from their employer.

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 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.

People fall into patterns when thinking about the privacy issues around apps [5]. By capturing these patterns explicitly as policies they can be enforced automatically. 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.

Companies have policies about how phones should be used by employees. With *bring your own device* schemes employees use their personal devices for work. IT departments in these companies may need to create policies so that the devices can be used securely with company information. Ensuring compliance is often left to the employees. By using an authorization logic we believe these policies can be enforced automatically. This reduces the burden on users to ensure compliance, and IT departments to check it.

2 Summary of second year work

In the thesis proposal we said we would primarily look at the role of app stores this year. We said that we 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 seemed to change as year went on. We implemented a system for creating app stores based on an AppPAL policy, but when we also gained access to real installation data. Using this data we started to look at what apps users pick from stores and the kinds of policies they apply. We started looking at greater detail at the distribution mechanisms and policies within app stores.

Some existing work continued on from the first year. The initial prototype of SecPAL became unweildy, wouldn't run on Android, and we found major bugs in the implementation. Consequently we reimplemented it, renamed it AppPAL, and have been optimizing and improving it through out the year. Other work was started from necessity. We wanted a means of systematically running tools over large numbers of apps and collecting the results; so we built one and have used the data collected from it in other aspects of the project.

In the next few sections we describe the work we have started in the second year.

2.1 Reimplemented AppPAL in Java.

The prototype started in the first year had several bugs, relating to policy checking, in it and became unweildy to fix. Furthermore because of lack of compiler support the implementation could not run on Android, making on device experimentation harder. We reimplemented 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.

The performance of the library when searching an assertion context is reasonable. Performance using synthetic benchmarks designed to test the search procedure is reasonable. The search procedure is at its slowest when having to handle large repeated delegations. We designed two synthetic benchmarks to check that the search procedure performed acceptably. 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 (though worse ones could of course be designed).

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

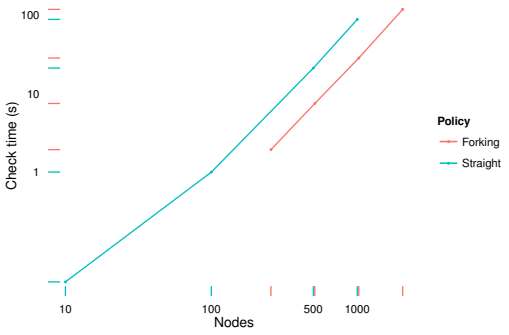


Figure 1: Benchmarking results on a Nexus 4 Android phone.

Cn 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 1). When describing hypothetical real policies, we typically only have a few delegations, and not long delegation chains, so we believe the performance will is acceptable. We believe that the AppPAL constraints will be the slowest part of checking a policy. Since they can execute arbitrary programs or include network requests their performance is independent of AppPAL. In practice the only ones we have used (so far) are permissions checks which are very fast. If slower checks are used in this may lead to policy design decisions to limit the number of constraints in policies. Since current practical applications checking 5 or 6 permissions per app are fast it is not something we should worry about until it becomes a problem, however.

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

As part of the AppPAL framework we imagined using results from static analysis tools as constraints within the language. To collect and store these results we 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). The SKB exports statements to AppPAL, and holds around 40,000 apps at the moment.

2.3 Explored existing app store privacy policies.

There are many different Android app stores available. We believed that each of the different stores would have different policies for kinds of apps, developers, and privacy conditions. We read the privacy, developer and user policies for four different app stores (Google Play, Amazon, Aptoide and Yandex) to compare them. In practice we 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.

Also some stores (with the exception of Google) 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 resigned by the store to be installed. Resigning is interesting because the trust in the authenticity of the app moves from the developer to the store. A summary of the different terms is shown in Figure 2.

2.4 Explored distribution mechanisms in existing stores.

When a user buys an app the app has to be sent to their devices somehow. Using an SSL proxy we 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. Various security flaws were found. Amazon's store was the only one to enable certificate pinning but this was only for the logon process, after which the SSL proxy worked fine. Google's store occasionally seemed to drop encryption when downloading APK files, allowing a listener to identify the apps someone used; though this seemed to stop happening a month later. Working from the network traffic and by reverse engineering the store we attempted to create an abstract description of the protocol used to buy apps (Figure 3).

2.5 User app installation behaviour

Mobile device users have policies about what data apps should be able to collected. Lin et al. [5] identified four different privacy policies when users think about apps; however they did not look at whether they enforced them in practice. Using AppPAL we implemented a simplification of the policies Lin et al.identified. The Carat project [6] collected app installation data from users who installed their energy tracking app. They shared this data with us in an anonymised form where we could see who had installed what provided we knew the package names of the apps they installed. Ignoring system apps, using the SKB we deannonimized 4,300 apps (5%) which accounted for 50% of all app installs. We selected 44,000 users for whom we knew 20 or more app installs.

Using the Carat data we measured the extent each user conformed with the privacy policies. We also took a list of known malware and **PUP!** (**PUP!**) software from McAfee to measure the extent of malware installations on Android. From this we tested whether users who enforce their privacy policies install less malware.

We found that very few users follow these policies, but a few users who do seem to be installing apps meeting these policies most of the time (Figure 4a). 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.

We found 1% of the users had a PUP or malicious app installed (Figure 4b). A user is 3 times more likely to have a PUP installed than malware. Only 9 users had both a PUP and malware installed. Users who were complying more than half the time with

	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. May request a refund for two hours after purchase.	No.	Upto 15 minutes after purchase. No for IAP.	Upto 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 bugfixes.	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	Marketting and optimizing Android.	Distribution, evaluation, modification, advertising, and creating derivatives for promotion.	Advertising.	Modification and re-selling.
Withdrawl from sale	Immediate.	10 days, or 5 days if for copywrite 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 unpredictible network use.	No offensive, pornography, illegal, IP infringing or gambling content.	No defects. No illegal, disrutpive, sexual, IP infinging, 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 2: Summary of conditions in different stores.

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 we found we could redownload the app on a different client. We 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 responce (seemingly derived from ? and U).
\$	Download token.
S'	Alternate store URL

Figure 3: Abstraction of the protocol used by Google's Play store to purchase an app.

the conservative or advanced policies complied with the malware or PUP policies fully (Figure 4c). This is significant (P-value < 0.05) and suggests that users who pick their apps carefully are less likely to experience malware.

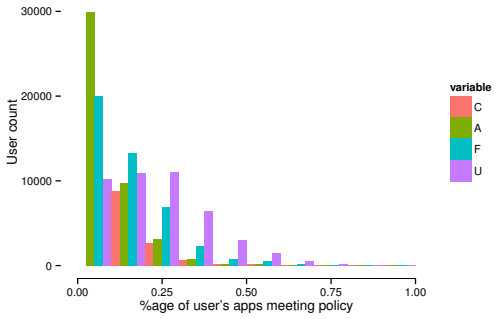
This work was presented as a poster at SOUPS [3], and we are aiming to publish it fully as part of a paper targetting the ESSoS conference (a draft of which is attached as an appendix).

3 Next directions

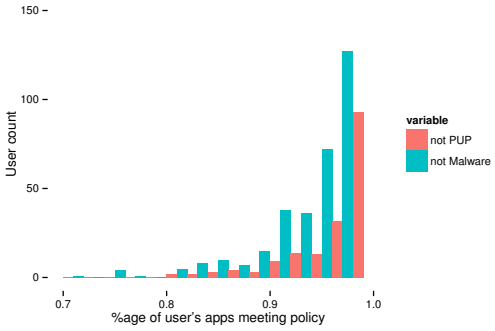
In the thesis proposal we suggested that in the third year we might look at advanced kinds of policies. We also wanted to look at what might happen when policies have to deal with updates, to policies and apps on the device. Some of the topics included:

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

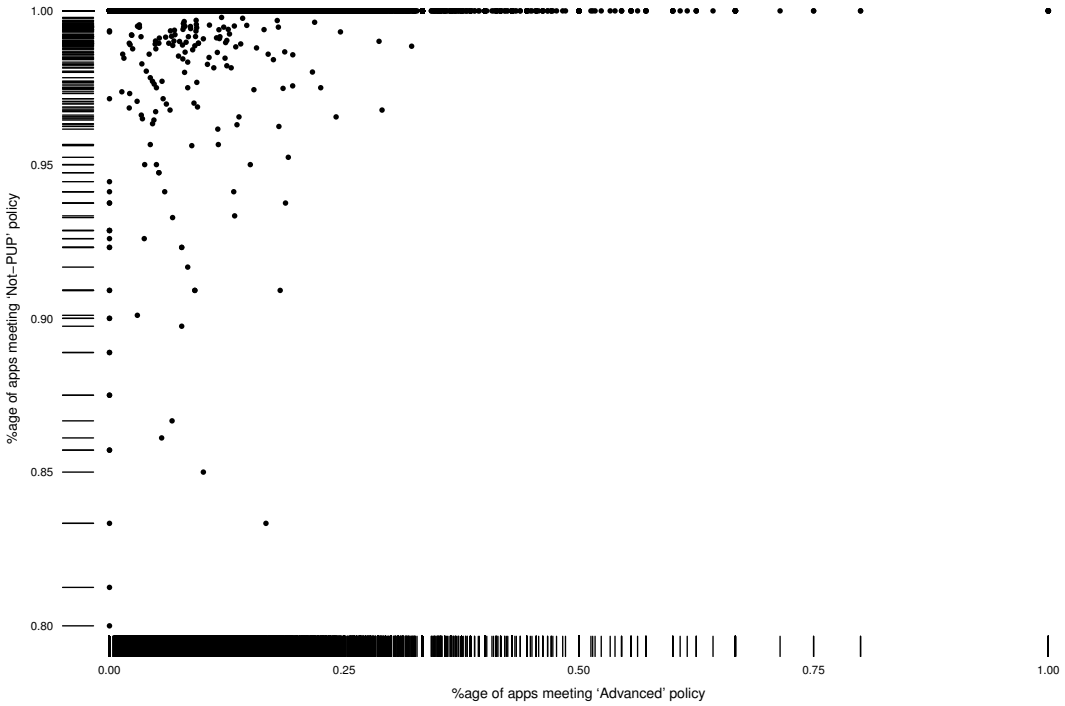
Some of these problems have become less interesting. App updates aren't interesting. Various different policies could be encoded in AppPAL for the precise behaviour but since most stores mandate accepting updates in the user agreement the point is moot. Policy modification likewise is not interesting. All apps will likely need to be rechecked



(a) User conformance to identified privacy policies.



(b) User malware installation rates.



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

Figure 4: Results from study on user app installation behaviour.

for compliance. Results from long-running static analysis tools can be stored by the SKB.

The app collusion problem is still interesting. Whilst it is concievable 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[1] 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. Further research here should continue along with the protocols for distributing AppPAL statements.

3.1 Knowledge distribution protocol

We have a means of enforcing policies and we have started looking at the kinds of policies users might want to use.

When describing AppPAL policies we 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 clearly 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 [2, 1] (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 a 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.

Consider the following scenario: Alice wants to install an app on her phone. As part of her installation policy she 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 really 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,

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

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.

Part of the continuing work should look at developing a strategy, and defining a protocol to share and acquire knowledge through AppPAL statements on demand.

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 we have learnt from user behaviour.

A natural source of these real world policies is corporate bring your own device (BYOD) policies, where employers describe restrictions on the useage of mobile devices in the workplace. The US National Institute of Standards and Technology (NIST) have published recomendations for mobile device security and BYOD in the workplace [9, 8]. 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.

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

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.

Deployment

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

Distribution

- Define protocol for knowledge acquisition.
- Describe implementation of protocol.

4. Evaluation

Case study

- Show how we can take a corporate policy and enforce it using AppPAL.
- Show the language is expressive enough to describe real policy scenarios.

- Show it works in *the real world*.

5. Related work

6. Future work

References

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

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

[3] 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.

[4] 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.

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

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

[7] 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.

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

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