Authorization Logic for App Policies

Second Year Report

Joseph Hallett

August 25, 2015

This report describes second year work of my PhD. I give a brief overview of the app installation policies and the AppPAL authorisation logic; before describing the work completed this year. I review what topics suggested in the thesis proposal. Work re-implementing AppPAL, exploring store and user policies and looking at protocols for app distribution is described. I describe what I expect to work on in the final year of my studies. Finally, I 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 [12], reviews and privacy concerns [9], 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 [10]. 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.

These 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 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 1). 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

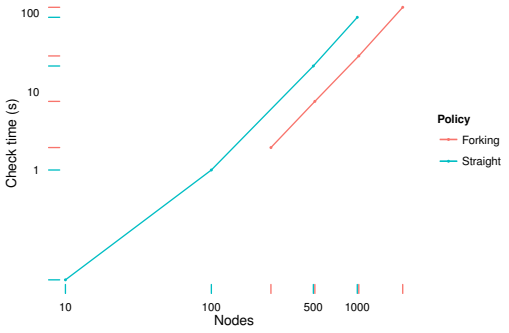


Figure 1: Bench-marking results on a Nexus 4 Android phone.

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 3.

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 2). The SKB can also emit statements from other principals where appropriate. For example in Figure 2 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 2: AppPAL output from the SKB.

```
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 3: 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 4.

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 4 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 4: Summary of conditions in different stores.

in abstract terms in Figure 5. 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 5: 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 6a). 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 6b). 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 6c). This is significant (P-value < 0.05) and suggests that users who pick their apps carefully are less likely to experience malware.

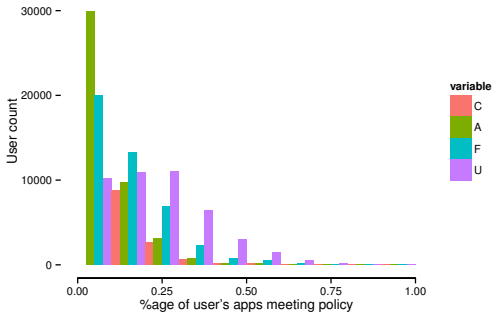
This work was presented as a poster at SOUPS [8], and I are aiming to publish it fully as part of a paper targeting the ESSoS conference (a draft of which is attached in an appendix).

3. Next directions

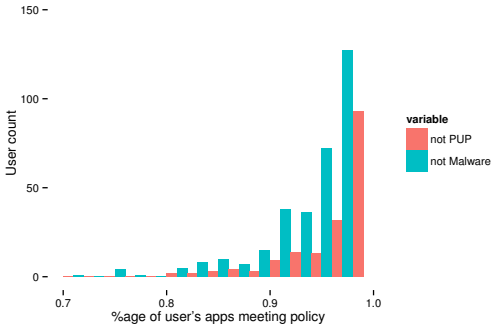
In the thesis proposal I suggested that in the third year I might look at advanced kinds of policies. I also wanted to look at what might happen when policies have to deal with updates, to policies and apps on the device. Some included:

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

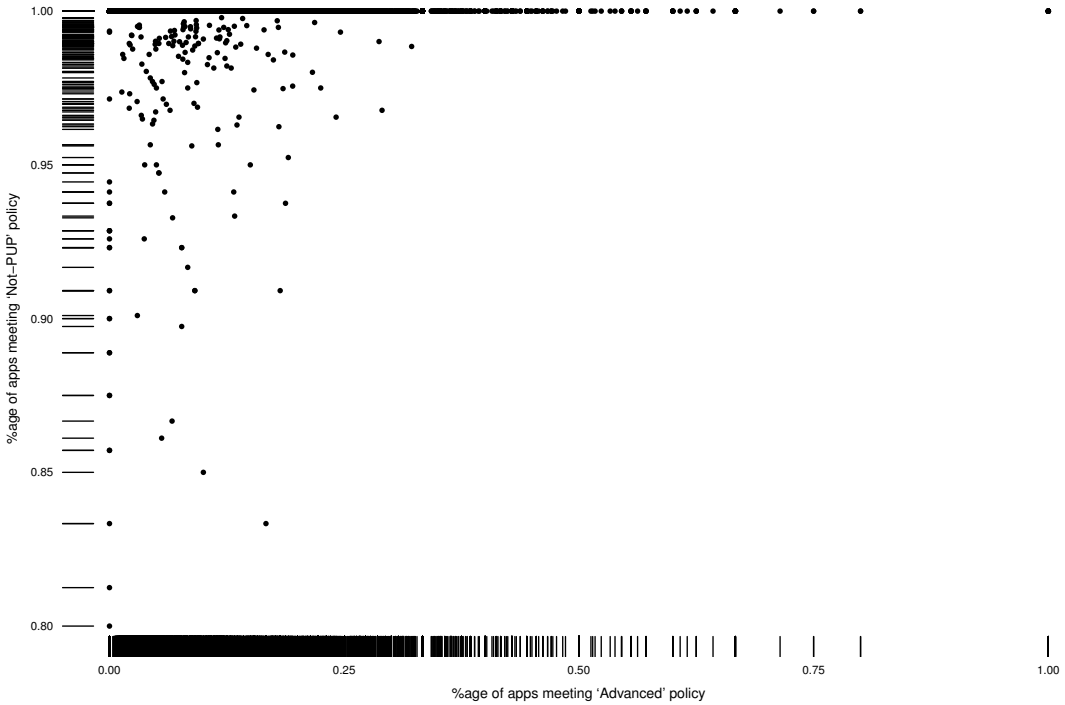
Some of these problems have become less interesting: for example app updates. Various different policies for updates could be encoded in AppPAL for the precise behaviour of an update. Since most stores mandate accepting updates in the user agreement and the default behaviour is to auto-install updates 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 6: 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[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. 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

[1] `'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 bring your own device (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[2] 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.

[2]Engineering Secure Software and Systems

AppPAL for Android
Capturing and Checking Mobile App Policies

Joseph Hallett and David Aspinall

University of Edinburgh

Abstract. Users must judge apps by the information shown to them by the store. Employers rely on employees enforcing company policies correctly on devices at their workplace. Some users take time to pick apps with care whilst others do not. Users feel frustrated when they realise what data the apps have access to. They want greater control over what data they give away but they do not want to spend time reading permissions lists. We present AppPAL: a policy language to enforce policies about apps. AppPAL Policies use statements from third parties and delegation relationships to give us a rigorous and flexible framework for enforcing and comparing and app policies; and the trust relationships surrounding them.

1 Introduction

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 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.

App stores give some information about their apps; descriptions of the app and screenshots as well as review scores. Android apps show a list of permissions when they're first installed. Soon new Android apps will display permissions requests when the app first tries to access sensitive data (such as contacts or location information). Users do not all understand how permissions relate to their device [16,36]. Ultimately the decision which apps to use and which permissions to grant must be made by the user.

Not all apps are suitable. A large amount of potentially unwanted programs (PUP) is being propagated for Android devices [37,35]. Employees are increasingly using their own phones for work (bring your own device (BYOD)). An employer may wish to restrict which apps their employees can use. The IT department may set a policy to prevent information leaks. Some users worry apps will misuse their personal data; such a user avoids apps which can access their location, or address book. They may apply their own personal security policy when downloading and running apps.

These policies can only be enforced by the users continuously making decisions guided by these policies when prompted about apps. This is error-prone.

Mistakes can be made. We believe this can be improved. An alternative would be to write the policy down and make the computer enforce it. To implement this we use a logic of authorization. The policy is written in the logic and enforced by checking the policy is satisfied.

We present AppPAL, an authorization logic for reasoning about apps. The language is an instantiation of SecPAL [5] with constraints and predicates that allow us to decide which apps to run or install. The language allows us to reason about apps using statements from third parties. The implementation allows us to enforce the policies on a device. We can express trust relationships amongst these parties; use constraints to do additional checks. This lets us enforce deeper and more complex policies than existing tools such as Kirin [13].

Using AppPAL we write policies for work and home, and decide which policy to use using a user's location, or the time of day:

```
"alice" says App isRunnable
  if "home-policy" isMetBy(App)
  where At("work") = false.

"alice" says App isRunnable
  if "work-policy" isMetBy(App)
  where BeforeHourOfDay("17") = true.
```

We can delegate policy specification to third parties or roles, and assign principals to those rolls:

```
"alice" says "it-department" can-say 0 "work-policy" isMetBy(App).
"alice" says "bob" can-act-as "it-department".
```

We can write policies specifying which permissions an app must or must not have by its app store categorization:

```
"alice" says App isRunnable
   if "permissions-policy" isMetBy(App).
"alice" says "permissions-policy" isMetBy(App)
   if App isAnApp
   where
      category(App, "Photography"),
      hasPermission(App, "LOCATION") = false,
      hasPermission(App, "CAMERA") = true.
```

Specifically our contribution is:

- Described a scenario where an employer has a policy they want to enforce for their employees (Section 2); and shown how the employer's policy could be implemented using AppPAL (Section 3) and installed on Alice's phone.
- Implemented the AppPAL language on Android and the JVM. We show how the language can describe properties of Android apps and Android security poilicies (Section 4) and how we can check policies (Subsection 4.1).
- Shown the need for policy tools by demonstrating the privacy paradox holds for user privacy policies with app installations (Section 5).

2 Enforcing a policy at work

An employee *Alice* works for her *em*ployer *Emma*. Emma allows Alice to use her personal phone as a work phone, but she has some specific concerns.

- Alice shouldn't run any apps that can track her movements. The testing labs are at a secret location and it mustn't be leaked.
- Apps should come from a reputable source, such as the Google Play Store.
- Emma uses an anti-virus (AV) program by McAfee. It should check all apps before they're installed.

To ensure this policy is met Alice promises to follow it. She might even sign a document promising never to break the rules within the policy. This is error-prone though—what if she makes a mistake or misses an app that breaks her policy? Emma's policy could be implemented using existing tools. The enterprise tool *Google's Device Policy for Android*[1] could configure Alice's device to disallow apps from outside the Google Play Store (and in the newest version of Android let Emma set the permissions of each app on an app by app basis [**?**]). AppPAL is designed to build on these tools, but make the trust and delegation relationships explicit and formalize the decision making process. Various tools such as AppGuard [2], Dr. Android & Mr. Hide [27] or AppFence [25] can control the permissions or data an app can get. These could be used to used to ensure no location data is ever obtained. Alternately other tools like Kirin [13], Flowdroid [19] or DroidSafe [21] could check that the locations are ever leaked to the web. Various anti-virus programs are available for Android—one could be installed on Alice's phone checking against McAfee's signatures.

Whilst we could implement Emma's policy using existing tools, it is a clumsy solution. They are not flexible: If Emma changes her policy or Alice changes jobs she needs to recheck and then to alter and remove the software on her phone accordingly. It isn't clear what an app must do to be run, or what checks have been done if it already running on the phone. The relationship between Alice (the user), Emma (the policy setter) and the tools Emma trusts to implement her policy isn't immediately apparent.

What happens when Alice goes home? Emma shouldn't be able to overly control what Alice does in her private life. Alice might not be allowed to use location tracking apps at work, but at home she might want to (to meet friends, track jogging routes or find restaurants for example). Some mobile OSs allow app permissions to be enabled and disable at run time. Can we enforce different policies at different times or locations?

Our research looks at the problem of picking software. Given there are some apps you want to install and run and others you do not want to, at least some of the time: how can you express your preferences in such a way that they can be enforced automatically? How can we translate policy documents from natural language into a machine checkable form? Furthermore how can we show the trust relationships used to make these decisions clearly and precisely?

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

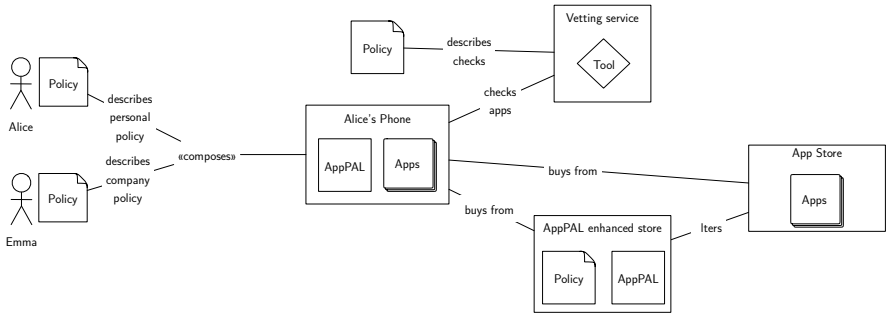


Fig. 1. Ecosystem of devices and stores with AppPAL.

We propose a change to the ecosystem, shown in Figure 1. People have policies which are enforced by AppPAL on their devices. The device can make use of vetting services which run tools to infer complex properties about apps. Users can buy from stores which ensure the only apps they see are the apps which meet their policies. Policies can be combined

3 Expressing policies in AppPAL

In Section 2 Alice and Emma's had policies they wanted to enforce but no means to do so. Instead of using several different tools to enforce Emma's policy disjointedly, we use an authorization logic. In Figure 2 we give an AppPAL policy implementing Emma's app concerns on Alice's phone.

AppPAL is an instantiation of SecPAL [5] for describing app policies. SecPAL is a logic of authorization for access control decisions in distributed systems. It has a clear and readable syntax, as well as rich mechanisms for delegation and constraints. SecPAL has already been used as a basis for other policy languages in areas such as privacy preferences [6] and data-sharing [1]. We present AppPAL as a new version of SecPAL, targeting apps on mobile devices.

In line 2 Alice gives Emma the ability to specify whether an `App` (a variable) `isRunnable` (a predicate). She allows her to delegate the decision further if she chooses (*`can-sayinf`*). Next in line 4 Emma specifies her concerns as policies to be met (the `isMetBy()` predicate that takes an app as its argument). If Emma can be convinced all these policies are met then he will say the `App isRunnable`. In line 10 and line 14 Emma specifies that an app meets the `reputable-policy` if the `App isReputable`; with "google−play" specified as the decider of what is buyable or not. This time Google is not allowed to delegate the decision further (*`can-say0`*). In other words Google is not allowed to specify Amazon as a supplier of apps as well. Google must say what is buyable directly for Emma to accept it. Emma specifies the "anti−virus−policy" in line 15. Here we use a constraint. When checking the policy the `mcAfeeVirusCheck` should be run on the `App`. Only if this returns `false` will the policy be met. To specify the "no−tracking−policy"

```
1  "alice" says "emma" can-say inf
2    App isRunnable.
3
4  "emma" says App isRunnable
5    if "no-tracking-policy" isMetBy(App)
         ,
6       "reputable-policy" isMetBy(App),
7       "anti-virus-policy" isMetBy(App).
8
9  "emma" says
10   "reputable-policy" isMetBy(App)
11      if App isReputable.
12
13 "emma" says "google-play" can-say 0
14   App isReputable.
15
16 "emma" says "anti-virus-policy"
17      isMetBy(App)
18   if App isAnApp
19   where
20     mcAfeeVirusCheck(App) = false.
21
22 "emma" says "no-location-permissions"
23   can-act-as "no-tracking-policy".
24
25 "emma" says
26   "no-location-permissions" isMetBy(App
27       )
28     if App isAnApp
29     where
30       hasPermission(App, "LOCATION"
31           )=false.
```

Fig. 2. AppPAL policy implementing Emma's security requirements

Emma says that the "no-location-permissions" rules implement the "no-tracking-policy" (line 21). Emma specifies this in line 24 by checking the app is missing two permissions.

Alice wants to install a new app (`com.facebook.katana`) on her phone. To meet Emma's policy the AppPAL policy checker needs to collect statements to show the app meets the `isRunnable` predicate. Specifically it needs:

- "emma" *says* "com.facebook.katana" `isAnApp.` A simple typing statement that can be generated for all apps as they are encountered. This helps keep the number of assertions in the policy low aiding readability.
- "google-play" *says* "com.facebook.katana" `isReputable.` Required to convince Emma the app came from a reputable source. It should be able to obtain this statement from the Play store as the app is available there.
- "emma" *says* "anti-virus-policy" `isMetBy("com.facebook.katana").` She can obtain this by running the AV program on her app.
- "emma" *says* "no-locations-permissions" `isMetBy("com.facebook.katana").` Needed to show the App meets Emma's no-tracking-policy. Emma will say this if after examining the app the location permissions are missing.

These last two statements require the checker to do some extra checks to satisfy the constraints. To get the third statement it must run the AV program on her app and check the result. The results from the AV program may change with time as it's signatures are updated; so the checker must re-run this check every time it wants to obtain the statement connected to the constraint. For the forth statement the checker needs to check the permissions of the app. It could do this by looking in the `MANIFEST.xml` inside the app itself, or through the Android package manager if running on device.

In this scenario we have imagined Alice wanting to check the apps as she installs them. Alternatively we could imagine Emma wanting a personalised app store where all apps sold meet her policy. With AppPAL this can be implemented by taking an existing store and selectively offering only the apps which will meet the user's policy. This gives us a *filtered store*. From an existing set of apps we produce a personalised store that meets a pre-defined policy.

4 AppPAL

AppPAL is an instantiation of SecPAL for Android app installation policies. It is implemented as a library for Android and Java. The parser is implemented using ANTLR4. Code and build instructions are available from Github[2].

The structure of an AppPAL assertion can be seen in Figure 3.

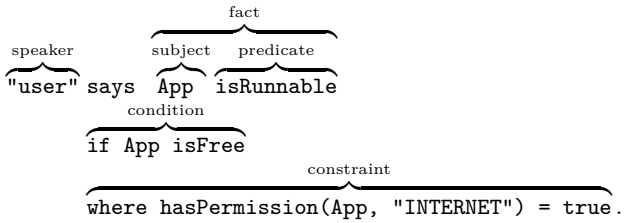


Fig. 3. Structure of an AppPAL assertion.

In the Becker et al.'s paper [5] they leave the choice of predicates, and constraints for their SecPAL open. With AppPAL we make explicit our predicates and how they relate to Android. Some of these predicates require arguments. AppPAL policies typically make use of the predicates and constraints in Table 1. Additional predicates can be created in the policy files, but adding or modifying constraints requires a code change[3]

Splitting the decision about whether an app is runnable into a series of policies that must be met gives us flexibility in how the decision is made. It allows us to describe multiple means of making the same decision, and provide backup routes when one means fails. Some static analysis tools are not quick to run. Even taking minutes to run a battery draining analysis can be undesirable. If a user wants to download an app quickly they may not be willing to wait to check that a policy is met.

[2] `https://github.com/bogwonch/libAppPAL`

[3] But a small one. The Android version of the `hasPermission` constraint, for example, uses the Android package manager to determine what permissions an app requests, but the Java version uses the Android platform tools. This required a code change to one function.

Name	Description
App `isRunnable`	Says an app can be run.
App `isInstallable`	Says an app can be installed.
App `isAnApp`	Tells AppPAL that an app exists.
Policy `isMetBy(App)`	Says a specific policy is met by an app. This is used to split policies into smaller components which can be reused and composed.
`hasPermission(App, Permission)`	Constraint for testing whether an app has been granted a permission.
`BeforeHourOfDay(time)`	Constraint used to test whether we're before an hour in the day.

Table 1. Typical AppPAL predicates and constraints

Earlier we described a *no-tracking-policy* to prevent a user's location being tracked. In Bob's policy we checked this by checking the permissions of the app. If the app couldn't get access to the GPS sensors (using the permissions) then it met this policy. Some apps may want to access this data, but may not leak it. We could use a taint analysis tool to detect this (e.g. Taintdroid [19]). Our policy now becomes:

```
"bob" says "no-locations-permissions"
  can-act-as "no-tracking-policy".

"bob" says "no-locations-permissions" isMetBy(App)
  if App isAnApp
  where
    hasPermission(App, "ACCESS_FINE_LOCATION") = false,
    hasPermission(App, "ACCESS_COARSE_LOCATION") = false.

"bob" says "location-taint-analysis"
  can-act-as "no-tracking-policy".

"bob" says "location-taint-analysis" isMetBy(App)
  if App isAnApp
  where
    taintDroidCheckLeak(App, "Location", "Internet") = false.
```

Sometimes we might want to use location data. For instance Bob might want to check that Alice is at her office. Bob might track Alice using a location tracking app. Provided the app only talks to Bob, and it uses SSL correctly (which Mallodroid can check for [14]) he is happy to relax the policy.

```
"bob" says "relaxed-no-tracking-policy" canActAs "no-tracking-policy".
"bob" says "relaxed-no-tracking-policy" isMetBy(App)
  if App hasCategory("tracking")
  where
    mallodroidSSLCheck(App) = false,
    connections(App) = "[https://bob.com]".
```

This gives us four different ways of satisfying the *no-tracking-policy*: with permissions, with taint analysis, with a relaxed version of the policy, or by Bob directly saying the app meets it. When we come to check the policy if any of these ways give us a positive result we can can stop our search.

4.1 Policy checking

Since AppPAL is an instantiation of SecPAL the policy checking rules are the same. We do not use Becker et al.'s DatalogC [30] based translation and evaluation algorithm, as no DatalogC library exists for Android. We have implemented the rules directly in Java. Pseudo-code is given in Figure 4. Like Becker et al.we make use of an assertion context to store known statements. We also store intermediate results to avoid re-computation. On a mobile device memory is at a premium. We would like to keep the context as small as possible. For some assertions (like `isAnApp`) we derive them by checking the arguments at evaluation time.

This gives us greater control of the evaluation and how the assertion context is created. For example, when checking the `isAnApp` predicate; we can fetch the assertion that the subject is an app based on the app in question. Similarly when we use a statement from *Emma* that *Google-Play can-say* whether an app is buyable; it is sensible to go fetch from the store whether the app is saleable and make Google say it then and there.

```
def evaluate(ac, rt, q, d)
  return rt[q, d] if rt.contains q, d
  p = cond(ac, rt, q, d)
  return (Proven, rt.update q, d, p)
      if p.isValid
  p = canSay_CanActAs(ac, rt, q, d)
  return (Proven, rt.update q, d, p)
      if p.isValid
  return (Failure, rt.update q, d,
      Failure)

def canSay_CanActAs(ac, rt, q, d)
  ac.constants.each do |c|
    if c.is_a :subject
      p = canActAs ac, rt, q, d
      return Proven if p.isValid
    elsif c.is_a :speaker
      p = canSay ac, rt, q d
      return Proven if p.isValid
  return Failure
```

```
def cond(ac, rt, q, d)
  ac.add q.fetch if q.isFetchable
  ac.assertions.each do |a|
    if (u = q.unify a.consequent) &&
       (a = u.substitution a).
            variables == none
      return checkConditions ac, rt,
          a, d
  return Failure

def checkConditions(ac, rt, a, d)
  getVarSubstitutions(a,ac.constants
      ).each do |s|
    sa = s.substitute a
    if sa.antecedents.all
        { |a| evaluate(ac, rt, a, d).
            isValid }
      p = evaluateC sa.constraint
      return Proven if p.isValid
  return Failure
```

Fig. 4. Partial-pseudocode for AppPAL evaluation.

4.2 Benchmarks

To test the performance of AppPAL we designed two policies that to compare worst case performance. Each policy was set up so that the entire policy space would have to be searched to find the result; giving the worst case performance. The *straight* policy gave a chain of delegations so that each user delegated to one additional user. The *forking* policy gave a tree of delegations so that each user delegated to two more users in a binary tree. A snippet of each policy is shown in Figure 5.

```
'1' says '2' can-say App
    isInstallable.
'2' says '3' can-say App
    isInstallable.
'3' says '4' can-say App
    isInstallable.
'4' says '5' can-say App
    isInstallable.
'5' says '6' can-say App
    isInstallable.
'6' says '7' can-say App
    isInstallable.
```

```
'1' says '2' can-say App
    isInstallable.
'1' says '3' can-say App
    isInstallable.
'2' says '4' can-say App
    isInstallable.
'2' says '5' can-say App
    isInstallable.
'3' says '6' can-say App
    isInstallable.
'3' says '7' can-say App
    isInstallable.
```

Fig. 5. Straight and forking policies used for benchmarking.

Using a Google Nexus 4 device each policy was extended until it was deep enough and the real-time measured for parsing each policy, and checking whether an app satisfied it. Results are shown in Figure 6. The benchmarks were also run on a more powerful desktop machine (a 2.6 GHz Intel Core i7) and are comparable and roughly 10 times quicker. Whilst the performance is exponential to the number of nodes to be traversed, it is not problematic. These are worst case performance times with unusually large delegation chains. We would expect most real device policies to have delegation chains of only a few levels, and certainly less than 10. If longer chains are required then the checking could be sent to a more powerful server, which can do them quicker than the mobile phone.

5 Demonstration

AppPAL can be used as a tool for exploring policy compliance in app installation data sets. The *privacy paradox* is that whilst people often have opinions about use of their personal data, they do not always follow through with their actions. Users seem to have opinions about apps [32] but are they picking apps which follow their policies? If the privacy paradox occurs here too then it suggests the need for tools to at least make users aware they're breaking their own policies.

Policy	Nodes	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

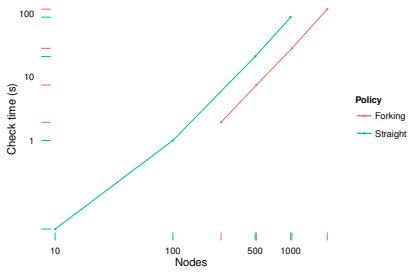


Fig. 6. Benchmarking results on a Nexus 4 Android phone.

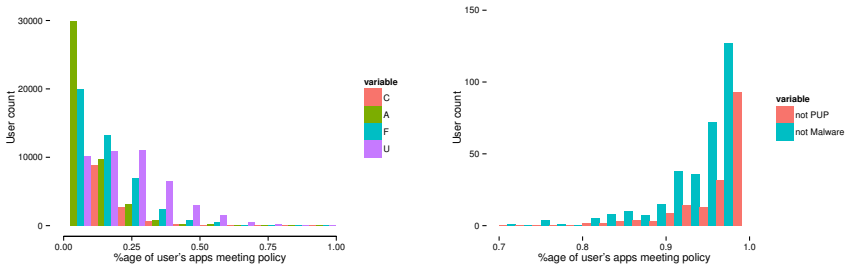
In a user study of 725 Android users, Lin et al.found four patterns that characterise user privacy preferences for apps [32]. The *Conservative* (C) users were uncomfortable allowing an app access to any personal data for any reason. The *Unconcerned* (U) users felt okay allowing access to most data for almost any reason. The *Advanced* (A) users were comfortable allowing apps access to location data but not if it was for advertising reasons. Opinions in the largest cluster, *Fencesitters* (F), varied but were broadly against collection of personal data for advertising purposes. We wrote AppPAL policies to describe each of these behaviours as increasing sets of permissions. These simplify privacy policies identified as by Lin et al.as we do not take into account the reason each app might have been collecting each permission. AppPAL can describe more complex policies however.

Policy	**C**	**A**	**F**	**U**
GET_ACCOUNTS	✗	✗	✗	✗
ACCESS_FINE_LOCATION	✗	✗	✗	
READ_CONTACT	✗	✗	✗	
READ_PHONE_STATE	✗	✗		
SEND_SMS	✗	✗		
ACCESS_COARSE_LOCATION	✗			

Like other vendors, McAfee classify malware into several categories. The *malicious* and *trojan* categories describe traditional malware. Other categories classify PUP such as aggressive adware. Using AppPAL we can write policies to differentiate between different kinds of malware, characterising users who allow dangerous apps or those who install merely “unsavoury” apps.

```
"user" says "mcafee" can-say
  "malware" isKindOf(App).
"mcafee" says "trojan" can-act-as "malware".
"mcafee" says "pup" can-act-as "malware".
```

We want to test how well policies capture user behaviour. Installation data was taken from a partially anonymized[4] database of installed apps captured by Carat [33]. By calculating the hash of known package names we see who installed what. The initial database has over 90,000 apps and 55,000 users. On average each user installed around 90 apps each; 4,300 apps have known names. Disregarding system apps (such as `com.android.vending`) and very common apps (Facebook, Dropbox, Whatsapp, and Twitter) we reduced the set to an average of 20 known apps per user. To see some variations in app type, we considered only the 44,000 users who had more than 20 known apps. Using this data, and the apps themselves taken from the Google Play Store and Android Observatory [3], we checked which apps satisfied which policies.



(a) Use of policies modelling user behaviour.

(b) Percentage of malware in installed apps for users installing some malicious apps.

Fig. 7. Policy compliance graphs.

Figure 7(a) shows that the very few users follow these policies, but a few users who do seem to be installing apps meeting these policies 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.

We found 1% of the users had a PUP or malicious app installed. Figure 7(b) shows that infection rates for PUPs and malware is small. 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 the conservative or advanced policies complied with the malware or PUP policies fully (Figure 8). This is significant (P-value < 0.05) and suggests that users who pick their apps carefully are less likely to experience malware.

[4] Users are replaced with incrementing numbers, app names are replaced with hashes to protect sensitive names.

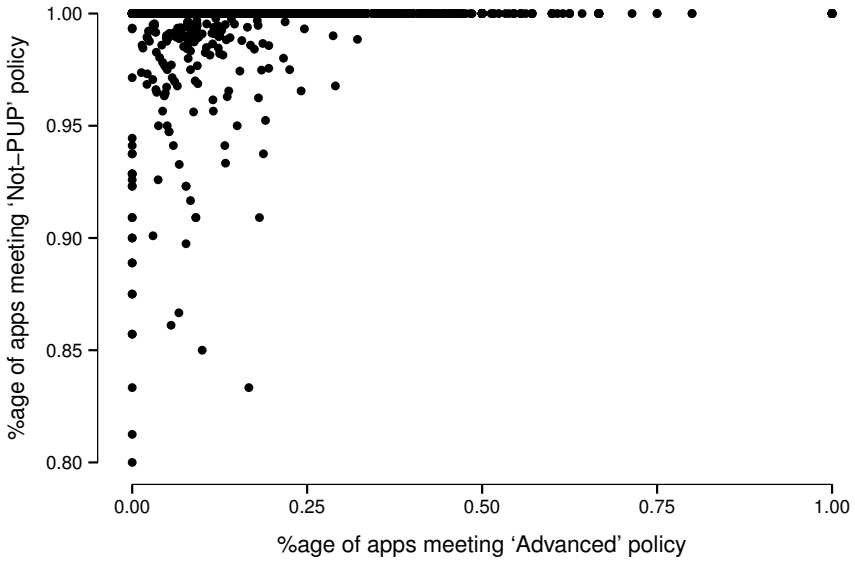


Fig. 8. Compliance with the advanced policy and non-PUP policy.

Most users seem to use apps irrespective of how uncomfortable they are with the permissions they request. A small set of users do seem to enforce these policies at least some of the time however. Exploring where this disconnect comes from is an avenue for future research. Do users not understand the relationship between apps and permissions [17]? Is enforcing them informally too difficult?

We claim that AppPAL can capture the differences in informal user policies. Using AppPAL we have written short policies describing user behaviour and used these to identify the users following them to varying degrees. Some limitations of our results include:

- We do not have the full user purchase history, and we can only find out about apps whose names match those in available databases. So a user may have apps installed that break the policy without us knowing.
- Recently downloaded apps used for experiment may not be the same version that users had, in particular, their permissions may differ. Permissions tend to increase in apps over time [38]; so a user may actually be more conservative than our analysis suggests.

6 Related work

Authorization logics have been successfuly used to enforce policies in several other domains. The earliest such logic, PolicyMaker [9], was general, if undecidable. Logics that followed like KeyNote [8] and SPKI/SDSI [11] looked at

public key infrastructure. The RT-languages [29,30,31] were designed for credential management. Cassandra [7] was used to model trust relationships in the british national health service.

SELinux is used to describe policies for Linux processes, and for access control (on top of the Linux discretionary controls). It was ported to Android [34] and is used in the implementation of the permissions system. Google also offer the *Device Policy for Android* app. This lets businesses to configure company owned devices to be trackable, remote lockable, set passwords and sync with their servers. It cannot be used to describe policies about apps, or describe trust relationships, however.

The SecPAL language was initially used for access control in distributed systems. We picked SecPAL as the basis for AppPAL because it was readable, extensible, and seemed to be a good fit for the problem we were trying to solve [24]. It has already been used to describe data usage policies [1] and inside Grid data systems [26]. Other work on SecPAL has added various features such as existential quantification [4] and ultimately becoming the DKAL family of policy languages [22,23]. Gruevich and Neeman showed that SecPAL was a subset of DKAL (minus the *can-act-as* statement). DKAL also contains more modalities than *says*, which lets policies describe actions principals carry out rather than just their oppinions. For example in AppPAL a user might *say* an app is installable if they would install it (`"user" says App isInstallable`) In DKAL they can describe the conditions that would force them to install it (`"user" installs App`). The distinction is that in AppPAL whilst the user thinks the app could be installed we do not know for sure whether the user has installed it. With the DKAL we can guarantee that the action was completed.

One tool, Kirin [13], also created a policy language and tool for enforcing app installation policies. Kirin's policies were concerned with preventing malware. Policy authors could specify combinations of permissions that should not appear together. For example an author might wish to stop malware sending premium rate text messages. To might implement this by restricting an app having both the `SEND_SMS` and `WRITE_SMS` permissions (Figure 9). Using this approach they found vulnerabilities in Android, but were ultimately limited by being restricted to permissions and broadcast events.

```
restrict permission [SEND_SMS] and permission [WRITE_SMS]

"user" says "no-write-send-sms" isMetBy(App)
  where hasPermission(App, "SEND_SMS") = false.
"user" says "no-write-send-sms" isMetBy(App)
  where hasPermission(App, "WRITE_SMS") = false.
```

Fig. 9. Kirin and AppPAL policies for stopping apps monetized by premium rate text messages.

This approach could help identify malware, but it is less suitable for detecting PUP. The behaviours and permissions PUP apps display aren't necessarily malicious. One user may consider apps which need in-app-purchases to play malware, but another may enjoy them. AppPAL tries to stop these PUP apps. Because we can use external checking tools which go further than permissions checks, our policies can be richer. By allowing delegation relationships we can understand the provenance and trust relationships in these rules.

There has been a great amount of work on developing app analysis tools for Android. Tools such as Stowaway [15] detect overprivileged apps. TaintDroid [12] and FlowDroid [19] can do taint and control flow analysis; sometimes even between app components. Other tools like QUIRE [10] can find privilege escalation attacks between entire apps. ScanDAL [28] and SCanDroid [20] help detect privacy leaks. Appscopy [18] searches for specific kinds of malware. Tools like DroidRanger [39] scan app markets for malicious apps. Many others exist checking and certifying other aspects of app behaviour.

7 Conclusions and further work

We have presented AppPAL: an authorization logic for describing app installation policies. The language is implemented in Java and runs on Android using a custom evaluation algorithm. This lets us to enforce app installation policies on Android devices. We have shown how the language can be used to describe an app installation policy; and given brief descriptions of how other policies might be described.

Further work is required to tightly integrate AppPAL into Android. One way to integrate AppPAL on Android would be as a *required checker*: a program that checks all apps before installation. Google uses this API to check for known malware and jailbreak apps. We would use AppPAL to check apps meet policies before installation. Unfortunately the API is protected and it would require the phone to be rooted to run there. Alternatively AppPAL could be integrated as a service to reconfigure app permissions. The latest version of Android[5] is moving to a more iOS like permissions model where permissions can be granted and revoked at any time. These will be manually configurable by the user through the settings app. We can imagine AppPAL working to reconfigure these settings (and set their initial grant or deny states) based on a user's policy, as well as the time of day or the user's location. A policy could deny pop-up notices while a user is driving for example.

Developing, and testing, policies for users is a key next step. Here we described a policy being specified by a company boss. For most end-users writing a policy in a formal language is too much work. Ad-blocking software works by users subscribing to filter policies written by experts[6]. We can imagine a simi-

[5] Called *Android M*.

[6] EasyList is a popular choice and the default in most ad-blocking software. They offer many different policies for specific use-cases however. `https://easylist.adblockplus.org/en/`

lar scheme working well for app installation policies. Users subscribe to different policies by experts (examples could include no tracking apps, nothing with adult content, no spammy in-app-purchase apps). Optionally they can customize them further.

We should also attempt to learn policies from existing users behavior. Given app usage data, from a project like Carat [33], we could identify security conscious users. If we can infer these users policies we may be able to describe new policies that the less technical users may want. Given a set of apps one user has already installed, we could learn policies about what their personal installation policy is. This may help stores show users apps they're more likely to buy, and users apps that already behave as they want.

AppPAL is a powerful language for describing app installation policies. It gives us a framework for describing and evaluating policies for Android apps. The work provides new ways for users to enforce their own rules about how apps should behave. Users policies can be enforced more reliably, and with less interaction; making apps more pleasant for everyone and helping to reduce user fatigue.

References

1. Aziz, B., Arenas, A., Wilson, M.: SecPAL4DSA. Cloud Computing and Intelligence Systems (2011)
2. Backes, M., Gerling, S., Hammer, C., Maffei, M.: AppGuard–Enforcing User Requirements on Android Apps. Tools and Algorithms for the Construction and Analysis of Systems (2013)
3. Barrera, D., Clark, J., McCarney, D., van Oorschot, P.C.: Understanding and improving app installation security mechanisms through empirical analysis of android. Security and Privacy in Smartphones and Mobile Devices pp. 81–92 (Oct 2012)
4. Becker, M.Y.: Secpal formalization and extensions. Tech. rep., Microsoft Research (2009)
5. Becker, M.Y., Fournet, C., Gordon, A.D.: SecPAL: Design and semantics of a decentralized authorization language. Computer Security Foundations (2006)
6. Becker, M.Y., Malkis, A., Bussard, L.: A framework for privacy preferences and data-handling policies. Tech. rep., Microsoft Research (2009)
7. Becker, M.Y., Sewell, P.: Cassandra: flexible trust management, applied to electronic health records. Computer Security Foundations pp. 139–154 (2004)
8. Blaze, M., Feigenbaum, J., Keromytis, A.D.: KeyNote: Trust Management for Public-Key Infrastructures. International Workshop on Security Protocols 1550(Chapter 9), 59–63 (Jan 1999)
9. Blaze, M., Feigenbaum, J., Lacy, J.: Decentralized trust management. Security and Privacy (1996)
10. Bugiel, S., Davi, L., Dmitrienko, A.: Towards taming privilege-escalation attacks on Android. Network and Distributed System Security Symposium (2012)
11. Ellison, C., Frantz, B., Lainpson, B., Rivest, R., Thomas, B.: RFC 2693: SPKI certificate theory. The Internet Society (1999)
12. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J.: TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. Operating Systems Design and Implementation (2010)

13. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. Computer and Communications Security pp. 235–245 (Nov 2009)
14. Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., Smith, M.: Why Eve and Mallory Love Android. ASIA Computer and Communications Security pp. 50–61 (Oct 2012)
15. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. Computer and Communications Security pp. 627–638 (Oct 2011)
16. Felt, A.P., Ha, E., Egelman, S., Haney, A., Chin, E., Wagner, D.: Android permissions: user attention, comprehension, and behavior. Symposium On Usable Privacy and Security p. 3 (Jul 2012)
17. Felt, A.P., Ha, E., Egelman, S., Haney, A., Chin, E., Wagner, D.: Android permissions: user attention, comprehension, and behavior. Symposium On Usable Privacy and Security p. 3 (Jul 2012)
18. Feng, Y., Anand, S., Dillig, I., Aiken, A.: Apposcopy: semantics-based detection of Android malware through static analysis. In: Foundations of Software Engineering. pp. 576–587. ACM Request Permissions, New York, New York, USA (Nov 2014)
19. Fritz, C., Arzt, S., Rasthofer, S.: Highly precise taint analysis for android applications. Tech. rep., Technische Univerität Darmstadt (2013)
20. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: SCanDroid: Automated security certification of Android applications. USENIX Security Symposium (2009)
21. Gordon, M.I., Kim, D., Perkins, J., Gilham, L., Nguyen, N.: Information-Flow Analysis of Android Applications in DroidSafe. Network and Distributed System Security Symposium (2015)
22. Gurevich, Y., Neeman, I.: DKAL: Distributed-Knowledge Authorization Language. Computer Security Foundations pp. 149–162 (2008)
23. Gurevich, Y., Neeman, I.: DKAL 2. Tech. Rep. MSR-TR-2009-11, Microsoft Research (Feb 2009)
24. Hallett, J., Aspinall, D.: Towards an authorization framework for app security checking. In: ESSoS Doctoral Symposium. University of Edinburgh (Feb 2014)
25. Hornyack, P., Han, S., Jung, J., Schechter, S.: These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In: Computer and Communications Security (2011)
26. Humphrey, M., Park, S.M., Feng, J., Beekwilder, N., Wasson, G., Hogg, J., LaMacchia, B., Dillaway, B.: Fine-Grained Access Control for GridFTP using SecPAL . Grid Computing (2007)
27. Jeon, J., Micinski, K.K., Vaughan, J.A., Fogel, A., Reddy, N., Foster, J.S., Millstein, T.: Dr. Android and Mr. Hide: fine-grained permissions in android applications. Security and Privacy in Smartphones and Mobile Devices pp. 3–14 (Oct 2012)
28. Kim, J., Yoon, Y., Yi, K., Shin, J., S Center: ScanDal: Static analyzer for detecting privacy leaks in android applications. ... on Security and Privacy (2012)
29. Li, N., Mitchell, J.C.: Design of a role-based trust-management framework. Security and Privacy (2002)
30. Li, N., Mitchell, J.C.: Datalog With Constraints. Practical Aspects of Declarative Languages 2562(Chapter 6), 58–73 (Jan 2003)
31. Li, N., Winsborough, W.H., Mitchell, J.C.: Distributed credential chain discovery in trust management. Journal of computer security (2003)
32. Lin, J., Liu, B., Sadeh, N., Hong, J.I.: Modeling Users' Mobile App Privacy Preferences. Symposium On Usable Privacy and Security (2014)
33. Oliner, A.J., Iyer, A.P., Stoica, I., Lagerspetz, E.: Carat: Collaborative energy diagnosis for mobile devices. In: Embedded Network Sensor Systems (2013)

34. Smalley, S., Craig, R.: Security Enhanced (SE) Android: Bringing Flexible MAC to Android. Network & Distributed System Security (2013)
35. Svajcer, V., McDonald, S.: Classifying PUAs in the Mobile Environment. sophos.com (Oct 2013)
36. Thompson, C., Johnson, M., Egelman, S., Wagner, D., King, J.: When it's better to ask forgiveness than get permission. In: the Ninth Symposium. p. 1. ACM Press, New York, New York, USA (2013)
37. Truong, H.T.T., Lagerspetz, E., Nurmi, P., Oliner, A.J., Tarkoma, S., Asokan, N., Bhattacharya, S.: The Company You Keep. World Wide Web pp. 39–50 (Apr 2014)
38. Wei, X., Gomez, L., Neamtiu, I., Faloutsos, M.: Permission evolution in the Android ecosystem. In: Anual Computer Security Applications Conference. pp. 31–40. ACM Request Permissions, New York, New York, USA (Dec 2012)
39. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. Network & Distributed System Security (2012)