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`

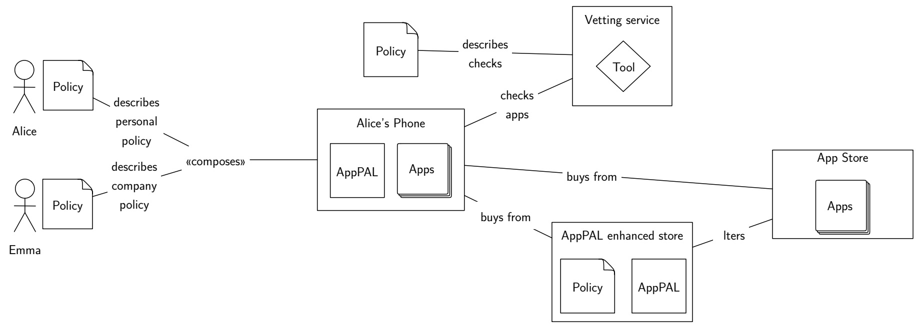


Fig. 1. Ecosystem of devices and stores with AppPAL.

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-say inf*). 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-say 0*). 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 "emma" says "anti-virus-policy"
     isMetBy(App)
16   if App isAnApp
17   where
18     mcAfeeVirusCheck(App) = false.
19
20 "emma" says "no-location-permissions"
21   can-act-as "no-tracking-policy".
22
23 "emma" says
24   "no-location-permissions" isMetBy(App
       )
25     if App isAnApp
26     where
27       hasPermission(App, "LOCATION"
           )=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.

$$\overbrace{\texttt{"user"}}^{\text{speaker}}\ \texttt{says}\ \overbrace{\overbrace{\texttt{App}}^{\text{subject}}\ \overbrace{\texttt{isRunnable}}^{\text{predicate}}}^{\text{fact}}$$

$$\overbrace{\texttt{if App isFree}}^{\text{condition}}$$

$$\overbrace{\texttt{where hasPermission(App, "INTERNET") = true}}^{\text{constraint}}.$$

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

Using the AppPAL library we wrote a benchmarking app that used policy that combined permissions checks and delegation relationships. The app measured the time it took to decide whether XX installed apps met the policy. We ran the app on X devices, and...

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.

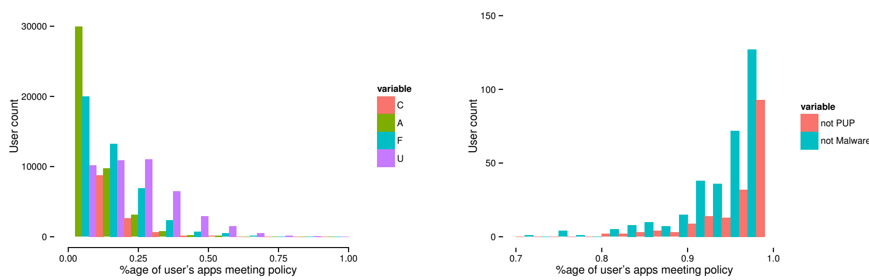
In a user study of 725 Android users, Lin et al.found four patterns that characterise user privacy preferences for apps [32]. The *Conservative* (C) users were uncomfortable allowing an app access to any personal data for any reason. The *Unconcerned* (U) users felt okay allowing access to most data 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. 5. Policy compliance graphs.

Figure 5(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 5(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 6). 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.

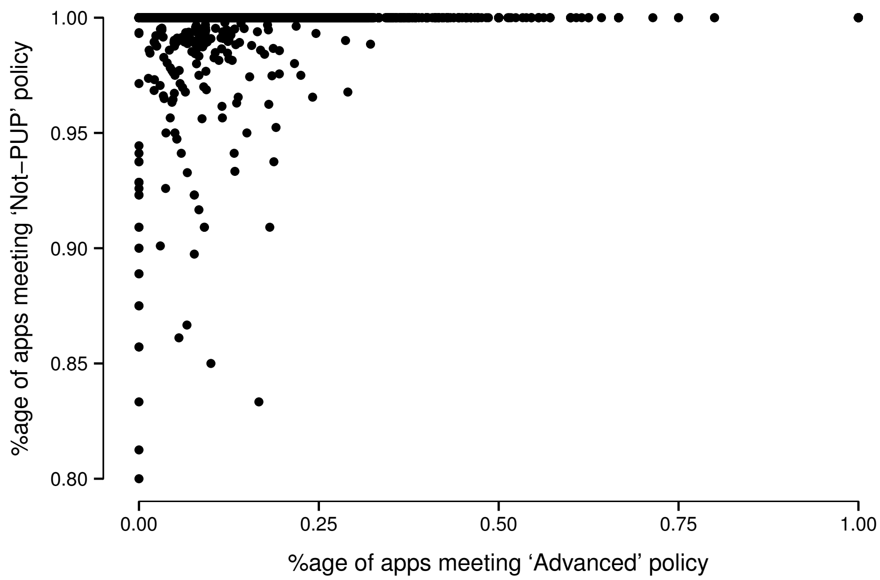


Fig. 6. Compliance with the advanced policy and non-PUP policy.

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 7). 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. 7. 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)