AppPAL for Android
Capturing and Checking Mobile App Policies

Joseph Hallett and David Aspinall

University of Edinburgh

Abstract. It can be difficult to find mobile apps that respect your security and privacy. At work, employers rely on employees enforcing company policies about apps at the workplace correctly. Users must judge apps by the information shown to them by the store. Only 17% of users pay attention to an apps permissions during installation [17] and most users do not understand how permissions relate to the capabilities of an app [30]. We present AppPAL: a policy language for Android to describe app installation policies. AppPAL goes beyond existing policy enforcement tools, such as Kirin [13], adding delegation relationships and the connecting to static analysis tools. We use AppPAL to explore what kinds of policies users use when installing apps, demonstrating the privacy paradox when comparing user's privacy views to their actual behaviour.

1 Introduction

Finding the right apps can be tricky. Users need to discover which are not going to abuse their data. This can be difficult as it isn't obvious how apps use the data each has access to. Consider a user attempting to buy a flashlight app. By searching the Play store the user is presented with a long list of apps. Clicking through each one they can find the permissions each requests but not the reasons why each was needed. They can see review scores from users but not experts who can run tools to check apps for problems and issues like SSL misconfigurations [15]. If they want to use the app at work will it break their employers rules for mobile usage?

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 apps will display permissions requests when the app first tries to access sensitive data (such as contacts or location information). Users do not understand how permissions relate to their device [17,41]. Ultimately the decision which apps to use and which permissions to grant must be made by the device user.

Some apps are highly undesirable. Many potentially unwanted programs (PUP) are being propagated for Android devices [42,40]. Employees are increasingly using their own phones for work. An employer may restrict which apps their employees can use. The IT department may set a policy—a series of rules describing what kinds of apps may be used and how—to prevent information leaks. Some users worry apps will misuse their personal data—sending their address book or location to an advertiser without their permission. Such a user avoids apps which can access their location, or address book. They may apply their own personal security policies when downloading and running apps.

These policies can only be enforced by the users continuously making the correct decision when prompted about apps. This is error-prone. An alternative is to write the policy down and make the computer enforce it. To implement this we use a logic of authorization—a language designed to express rules about permissible actions. The policy is written in the logic and enforced by checking the policy is satisfied.

We present AppPAL, an instantiation of Becker et al.'s SecPAL [5] with constraints[1] 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, such as using static analysis results. This lets us enforce more complex policies than existing tools such as Kirin [14] which are limited to permissions checks.

A user, Alice, may have rules she has to follow when using apps for work and her own policies when using apps at home in her private life. Using AppPAL we can write for work and home, and decide which policy to enforce 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 roles:

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

There has been a great amount of work on developing app analysis tools for Android. Tools such as Stowaway [16] detect over-privileged apps. TaintDroid [12] and FlowDroid [19] can do taint and control flow analysis; sometimes

[1] A constraint makes use of information checkable using information external to the language, such as the time of day or output of a static analysis tool

even between app components. Other tools like QUIRE [10] can find privilege escalation attacks between entire apps. ScanDAL [31] and SCanDroid [20] help detect privacy leaks. Appscopy [18] searches for specific kinds of malware. Tools like DroidRanger [44] scan app markets for malicious apps. Many others exist checking and certifying other aspects of app behaviour.

AppPAL can act as a *"glue"* between static analysis tools and the app installation policies device owner's are trying to enforce. For example a user might not want to install apps with SSL errors (which Mallodroid can check for [15]) or malicious apps as determined by their anti-virus. Using AppPAL we can combine tools for checking apps to implement the user's policies.

```
"alice" says App isInstallable
  if App isAnApp
  where mallodroidCheck(App) = true,
       avCheck(App) = true.
```

In this paper we have:

- 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.
- Described our implementation of the AppPAL language as a library for Android and the JVM. We show how the language can describe app properties and Android security policies (Section 4) as well as how to 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 employer *Emma*. Emma allows Alice to use her personal phone as a work phone but 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—what if she makes a mistake or misses an app that breaks her policy? Alternately Emma's policy could be partially enforced using existing tools. *Google's Device Policy for Android* [22] could configure Alice's device to disallow apps from outside the Google Play Store and let Emma set the permissions granted to each app [36].

AppPAL builds on these tools, but make the trust and delegation relationships explicit and provide a rigorous and extensible means for deciding whether an app meets a policy. Various tools such as AppGuard [2], Dr. Android & Mr. Hide [29] or AppFence [27] 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 [14], Flowdroid [19] or DroidSafe [23] could check that the locations are never 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 her apps and then alter or remove the software on her phone to ensure compliance. It isn't clear what an app must do to be run, or what checks have been done if it is 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, such as iOS and the latest version of Android, allow app permissions to be enabled and disable at run time. Can we enforce different policies at different times or locations?

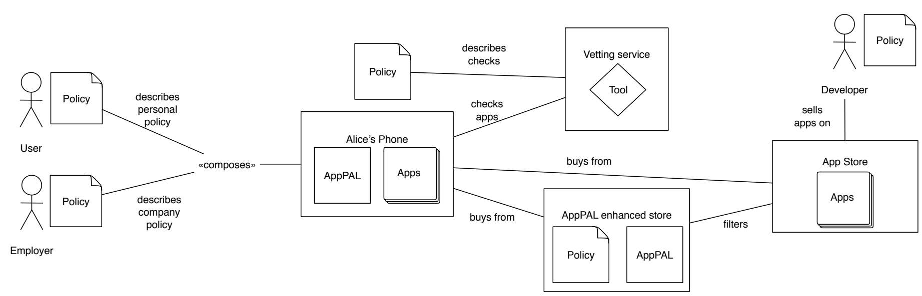


Fig. 1. Ecosystem of devices and stores with AppPAL.

We propose a change to the mobile ecosystem, shown in Figure 1. People have policies which are enforced by AppPAL on their devices. They can be composed with policies from employers or others to create enhanced devices that ensure apps meet the policies of their owners. The device can make use of vetting services which run tools to infer complex properties about apps. Users can buy from enhanced stores which ensure the only apps they sell are the apps which meet the explicitly specified store policies. Developers could decide which stores to sell their apps in on the basis of policies about stores.

3 Expressing policies in AppPAL

In Section 2 Alice and Emma 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 could use an authorization logic. In Figure 2 we give an AppPAL policy implementing Emma's app concerns on Alice's phone.

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 modified form of SecPAL, targeting apps on mobile devices.

```
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
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

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 she 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*). 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" 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 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 it is running on a device.

In this scenario we have imagined Alice wanting to check the apps as she installs them. We could also 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* which, from an existing set of apps, we get a personalised store that only sells apps that meet a policy.

4 AppPAL

AppPAL is implemented as a library for Android and Java. The parser is implemented using ANTLR4. The structure of an AppPAL is inherited from SecPAL [5] can be seen in Figure 3.

In SecPAL the precise nature of predicates and constraints is left open. In instantiating SecPAL, AppPAL makes the predicates and constraints explicit. AppPAL policies can make use of the predicates and constraints in Table 1. Additional predicates can be created in the policy files, however constraints are implemented individually. For example on Android the `hasPermission` constraint uses the Android package manager to check what permissions an app requests, but the Java version uses the Android platform tools to check.

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

$$\overbrace{\texttt{'user'}}^{\text{speaker}}\ \texttt{says}\ \overbrace{\overbrace{\texttt{App}}^{\text{subject}}\ \overbrace{\texttt{isRunnable}}^{\text{predicate}}}^{\text{fact}}$$
$$\overbrace{\texttt{if App isFree}}^{\text{condition}}$$
$$\overbrace{\texttt{where hasPermission(App, 'INTERNET') = true}}^{\text{constraint}}.$$

⟨Assertion⟩ ≔ ⟨E⟩ **says** ⟨Fact⟩
(**if** (⟨Fact⟩,)+)?**.**
(**where** ⟨Constraint⟩)?
⟨Fact⟩ ≔ ⟨E⟩ (**isRunable** | ...)
| ⟨E⟩ **can-say inf**? ⟨Fact⟩
| ⟨E⟩ **can-act-as** ⟨E⟩
⟨E⟩ ≔ **Variable** | **'constant'**

Fig. 3. Structure and simplified grammar of an AppPAL assertion.

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)	Used to split policies into smaller components.
`hasPermission`(App, Permission)	Constraint to check if an app has a permission.
`beforeHourOfDay`(time)	Constraint used to check the time.
Tool`Check`(App, Property)	Constraint to run an analysis tool on an app.

Table 1. AppPAL predicates and constraints.

routes when one 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.

In Section 2 and Section 3 we described a *no-tracking-policy* to prevent a user's location being leaked. In Emma's policy we checked this using the app's permissions. 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. FlowDroid [19]). Our policy now becomes:

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

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

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

"emma" says "location-taint-analysis" isMetBy(App)
  if App isAnApp
  where
```

```
  flowDroidCheck(App, "Location", "Internet") = false.
```

Sometimes we might want to use location data. For instance Emma might want to check that Alice is at her office. Emma might track Alice using a location tracking app. Provided the app only talks to Emma, and it uses SSL correctly (which Mallodroid can check for [15]) she is happy to relax the policy.

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

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 Emma 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 stop our search.

4.1 Policy checking

AppPAL has the same policy checking rules as SecPAL [5]. AppPAL uses an assertion context of known facts and rules We do not use Becker et al.'s DatalogC [33] based checking 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 and rules, as well as facts that are deduced during checking. On a mobile device memory is at a premium. We would like to keep the assertion 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.

4.2 Benchmarks

We envisage AppPAL running on a mobile phone, checking apps when they are installed. Since policy checks may involve inspecting many rules and constraints one may ask whether the checking will be acceptably fast. Downloading and installing an app takes about 30 seconds on a typical Android phone over wifi. If checking a policy delays this even further a user may become annoyed and disable AppPAL.

The policy checking procedure is at its slowest when having to delegate repeatedly; the depth of the delegation tree is the biggest factor for slowing the search. Synthetic benchmarks were created to check that the checking procedure performed acceptably. Each benchmark consisted of a chain of delegations. The *1 to 1* benchmark consists of a repeated delegation between all of the principals. In the *1 to 2* benchmark each principal delegated to 2 others and in the *1 to 3*

```
def evaluate(ac, rt, q, d)
  return rt[q, d] if rt.contains q, d
  p = cond(ac, rt, q, d)
  if p.isValid then
    return (Proven, rt.update q, d, p)
  p = canSay_CanActAs(ac, rt, q, d)
  if p.isValid then
    return (Proven, rt.update q, d, p)
  else
    return (Failure, rt.update q, d, 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.sub a).variables == none
      return checkConditions ac, rt, a, d
  return 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 checkConditions(ac, rt, a, d)
  getVarSubs(a,ac.constants).each do |s|
    sa = s.sub 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.

benchmark each principal delegated to 3 others. These benchmarks are reasonable as they model the slowest kinds of policies to evaluate—though worse ones could be designed by delegating even more.

For each benchmark we controlled the number of principals in the policy file: as the number of principals increased so did the size of the policy. The results are shown in Figure 5. We have only used a few delegations per decision when describing hypothetical user policies. We believe the policy checking performance of AppPAL is acceptable as unless a policy consists of hundreds of delegating principals the overhead of checking an AppPAL policy is negligable.

Delegations	Principals	Time (s)
1 to 1	10	0.01
1 to 1	100	1.00
1 to 1	500	20.90
1 to 1	1000	88.73
1 to 2	10	0.01
1 to 2	100	0.43
1 to 2	500	7.36
1 to 2	1000	27.47
1 to 3	10	0.01
1 to 3	100	0.24
1 to 3	500	3.99
1 to 3	1000	15.28

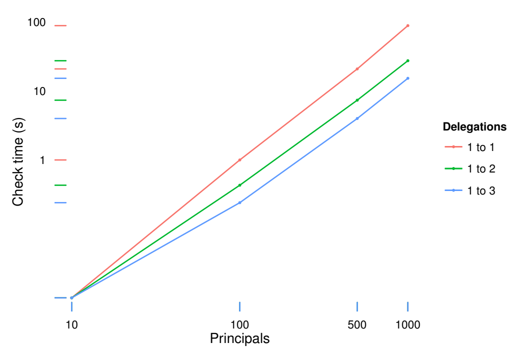


Fig. 5. Benchmarking results on a Nexus 4 Android phone.

5 Measuring Policy Compliance

Throughout we have asserted that user's have policies and that there is a need for policy enforcement tools. Corporate mobile security bring your own device (BYOD) policies have started appearing and NIST have issued recommendations for writing them [37,39]. In a study of 725 Android users, Lin et al. found four patterns that characterise user privacy preferences for apps [34]. Using app installation data from Carat [35] we used AppPAL to find the apps satisfying each policy Lin et al. identified and measured the extent each user was following them.

Lin et al. identified four types of user. 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 the 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.

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

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 and those who install poor quality ones.

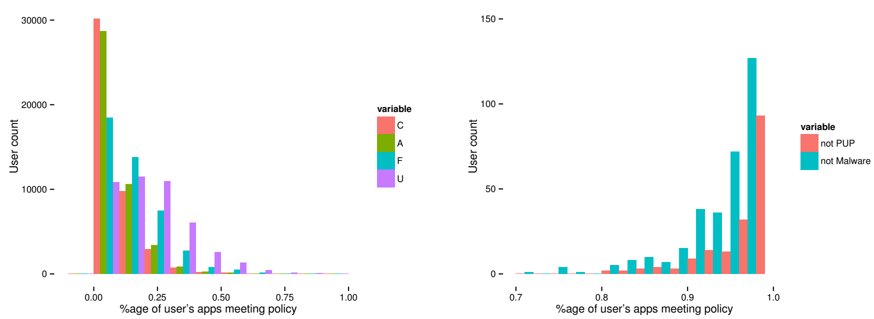
```
"user" says "mcafee" can-say
  "malware" isKindOf(App).
"mcafee" says "trojan" can-act-as "malware".
"mcafee" says "pup" can-act-as "malware".
```

If a user is enforcing a privacy policy we might also expect them to install less malware. We can check this by using AppPAL policies to measure the number of malwares each user had installed.

We want to test how well policies capture user behaviour. Installation data was taken from a partially anonymized[2] database of installed apps captured

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

by Carat [35]. By calculating the hashes 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.



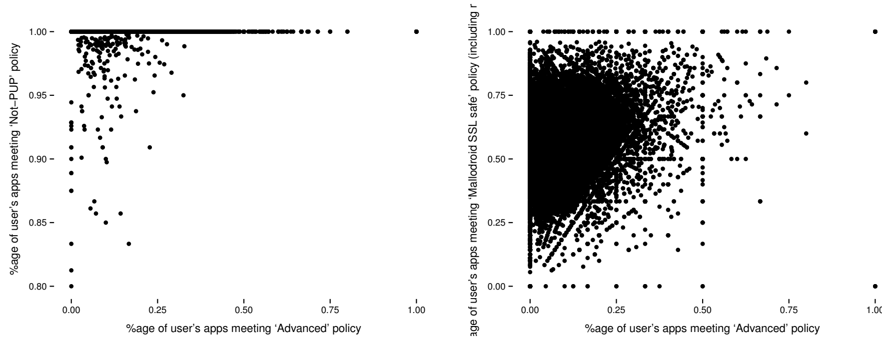
(a) Use of policies modelling user behaviour.

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

Fig. 6. Policy compliance graphs.

Figure 6(a) shows that very few users follow Lin et al.'s policies most of the time. Whilst the AppPAL policy we used was a simplified version of Lin et al.'s policy, it suggests that there is a disconnect between users privacy preferences and their behaviour (often referred to as the *privacy paradox*). A few users, however, did seem to be installing apps meeting these policies most of the time. This suggests that while users may have privacy preferences the majority are not attempting to enforce them. Policy enforcement tools, like AppPAL, can help users enforce their own policies which they cannot do easily using the current means available to them.

We found 1% of the users had a PUP or malicious app installed. Figure 6(b) shows that infection rates for PUPs and malware is low; though a user is 3 times more likely to have a PUP installed than malware. Users who were complying more than half the time with the conservative or advanced policies complied with the malware or PUP policies fully (Figure 7(a)). This suggests that policy enforcement is worthwhile: users who can enforce policies about their apps experience less malware.



(a) Advanced and non-PUP policies. (b) Advanced and Mallodroid policies.

Fig. 7. Compliance with the advanced policy and the non-PUP and SSL policies.

The *MalloDroid* tool [15] can scan apps for SSL misconfigurations. SSL misconfigurations are dangerous as they can undermine any privacy guarantees that SSL/TLS usage gives. We set up AppPAL to use Mallodroid results as a constraint and measured the percentage of apps each user had installed that did not have issues or suspected issues when scanned with Mallodroid. Users who were complying with the advanced policy were no better at avoiding apps with SSL errors than any other users (Figure 7(b)).

Some limitations of this investigation 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 [43]; so a user may be more conservative than our analysis suggests.
- The AppPAL policies we used are a simplification of the identified privacy preferences. User's could be following policies more nuanced than our implementation. We believe our approximation is valid, however, as users will not know the precise reason an app uses a permission, and may decline to install it anyway. Our policies could be made more precise by incorporating the app category, and allowing apps to have a permission when it is appropriate (i.e. a text messaging client will need the `SEND_SMS` permission to run).

6 Related work

Authorization logics have been successfully 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 [32] 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 [38] and is used in the implementation of the permissions system. SELinux describes the capabilities (in terms of system calls and file access) of processes, it cannot describe app installation policies or delegation relationships. Google also offer the *Device Policy for Android* app. This lets businesses 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 is designed for access control in distributed systems. We picked SecPAL as the basis for AppPAL because it is readable, extensible, and is a good for for the mobile ecosystem setting [26]. It has also been used to describe data usage policies [1] and inside Grid data systems [28]. Other work on SecPAL has added various features such as existential quantification [4] and the DKAL family of policy languages [24,25]. DKAL contains more modalities than *says*, which lets policies describe actions principals carry out rather than just their opinions. 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`). With DKAL we can guarantee that the action was completed, whereas in AppPAL we do not know if the user actually installed a particular app. We chose to use SecPAL as the basis for AppPAL as we did not need the extra features DKAL added to express app installation policies. If, in future work, we need additional modalities AppPAL could be extended to support additional DKAL features as SecPAL has been shown to be a subset of the DKAL language.

Kirin [14] is a policy language and tool for enforcing app installation policies to prevent malware. Policy authors can specify combinations of permissions and broadcast events that should not appear together. For example to stop malware sending premium rate text messages, we prevent an app having both the `SEND_SMS` and `WRITE_SMS` permissions.

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

By analyzing apps which broke their policies Enck et al. found vulnerabilities in Android, but were ultimately limited by being restricted to permissions and broadcast events.

This approach could help identify malware, but it is less suitable for detecting PUPS. The behaviours and permissions PUP displays aren't necessarily malicious. One user may not want apps which need in-app-purchases to play, but another may enjoy them. With Kirin we are restricted to permitting or allowing apps. AppPAL can describe more scenarios than just permit or allow, and use more app information than just permissions, such as constraints and static analysis results. By allowing delegation relationships we can understand the provenance and trust relationships in these rules.

7 Conclusions and further work

We have presented AppPAL: an authorization logic for describing app installation policies primarily designed to help achieve security and privacy objectives but which can also *lock down* devices in other ways. 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. Using app installation data and user privacy policies expressed using AppPAL we showed how the privacy paradox can be seen in the Android ecosystem. This motivates the need for tools

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 the required checker 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. Alternatively, AppPAL could be integrated as a service to reconfigure app permissions. Android Marshmallow has an 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 notifications while a user is driving, for example, by checking if they are using Android Auto [21] (an app to interact with a car's center console) or moving along a road at high speed.

Developing, and testing, policies for users is a key next step. Here we described a policy being specified by a user's employer. 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. EasyList is a popular choice and they offer many different policies for specific use-cases[3]. We can imagine a similar 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 might attempt to learn policies from existing users behavior. Given app usage data, from a project like Carat [35], 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.

[3] `https://easylist.adblockplus.org/en/`

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 7795(Chapter 39), 543–548 (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 pp. 164–173 (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.: Mitigating Android software misuse before it happens. Tech. Rep. NAS-TR-0094-2008 (Sep 2008)
14. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. Computer and Communications Security pp. 235–245 (Nov 2009)
15. 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)
16. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. Computer and Communications Security pp. 627–638 (Oct 2011)
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. Google: Android Auto

22. Google: Google Apps Device Policy
23. 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)
24. Gurevich, Y., Neeman, I.: DKAL: Distributed-Knowledge Authorization Language. Computer Security Foundations pp. 149–162 (2008)
25. Gurevich, Y., Neeman, I.: DKAL 2. Tech. Rep. MSR-TR-2009-11, Microsoft Research (Feb 2009)
26. Hallett, J., Aspinall, D.: Towards an authorization framework for app security checking. In: ESSoS Doctoral Symposium. University of Edinburgh (Feb 2014)
27. 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)
28. 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)
29. 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)
30. Kelley, P.G., Consolvo, S., Cranor, L.F., Jung, J., Sadeh, N., Wetherall, D.: A Conundrum of Permissions. Useable Security 7398(Chapter 6), 68–79 (Feb 2012)
31. 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)
32. Li, N., Mitchell, J.C.: Design of a role-based trust-management framework. Security and Privacy pp. 114–130 (2002)
33. Li, N., Mitchell, J.C.: Datalog With Constraints. Practical Aspects of Declarative Languages 2562(Chapter 6), 58–73 (Jan 2003)
34. Lin, J., Liu, B., Sadeh, N., Hong, J.I.: Modeling Users' Mobile App Privacy Preferences. Symposium On Usable Privacy and Security (2014)
35. Oliner, A.J., Iyer, A.P., Stoica, I., Lagerspetz, E.: Carat: Collaborative energy diagnosis for mobile devices. In: Embedded Network Sensor Systems (2013)
36. Poiesz, B.: Android M Permissions. In: Google I/O (2015)
37. Scarfone, K., Hoffman, P., Souppaya, M.: NIST Special Publication 800-46: Guide to Enterprise Telework and Remote Access Security (Jun 2009)
38. Smalley, S., Craig, R.: Security Enhanced (SE) Android: Bringing Flexible MAC to Android. Network & Distributed System Security (2013)
39. Souppaya, M., Scarfone, K.: NIST Special Publication 800-124: Guidelines for Managing the Security of Mobile Devices in the Enterprise (Jun 2013)
40. Svajcer, V., McDonald, S.: Classifying PUAs in the Mobile Environment. sophos.com (Oct 2013)
41. 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)
42. 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)
43. 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)

44. 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)