AppPAL for Android

Capturing and Checking Mobile App Policies

Joseph Hallett David Aspinall

June 8, 2015

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 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.

App stores give some information about their apps; such as permissions, descriptions of the app and screenshots as well as review scores. Android apps currently show a confusing 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 [15, 35]. 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 software (PUS) is being propagated for Android devices [36, 34]. Employees are increasingly using their own phones for work (bring your own device or 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 [11].

Using AppPAL we can say an app is installable:

```
"Alice" says "com.rovio.angrybirds" isInstallable.
```

We can delegate decisions to others:

```
"Alice" says "Bob" can-say 0 App isCool.
```

We can run analysis tools, and solve constraints:

```
"Claire" says App isMalware
  if App isAnApp
  where
    virusScanner(App) = true.
```

Constraints allow up to express facts that are true at some times but not others:

```
"Emma" says "com.facebook.katana" isRunnable
  where timeOfDay() > 1700.
```

Specifically we have:

- Described a scenario where an employer has a policy they want to enforce for their employees (Section 2.) We show how the trust relationships in a system can be difficult to see; and how AppPAL makes the trust between users explicit.
- Shown how the employer's policy could be implemented using AppPAL (Section 3) and installed on Alice's phone. We have described what AppPAL would check for an app to meet this policy and how these statements might be collected.
- Implemented the AppPAL language on Android and the JVM. We show how the language can describe properties of Android apps and Android security poilicies (Subsection 4.1), how we evaluate it (Subsection 4.2), and some idiomatic policies (Subsection 4.3).
- When things go wrong we also need to attribute blame. Authorization logic can help make trust relationships explicit. We show a recent example of a broken trust relationship (Subsection 4.1) and how our authorization logic makes explicit where the breach of trust is.

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 soon in the newest version of Android let Emma set the permissions of each app on an app by app basis). Various tools such as AppGuard [2], Dr. Android & Mr. Hide [25] or AppFence [23] 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 [11], Flowdroid [17] or DroidSafe [19] 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`

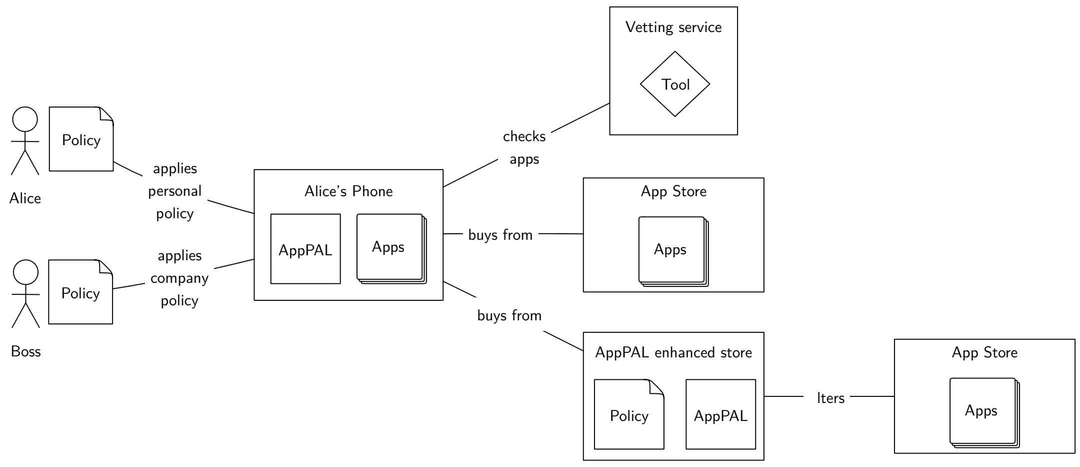

Figure 1: Ecosystem of devices and stores with AppPAL.

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.

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 14. Here we use a constraint. When

```
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.
```

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

Figure 2: AppPAL policy implementing Emma's security requirements

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 20). Emma specifies this in line 23 by checking the app is missing two permissions.

Say 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.

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

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

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

Figure 3: Structure of an AppPAL assertion.

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 implemented as a library for Android and Java. The parser is implemented using ANTLR4. Code and build instructions are available from Github[2].

4.1 The Language

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

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. We add a light typing system to help ensure these predicates have the correct arguments.

Specifically we define the following predicates:

`App isRunnable` Used to indicate that an App meets the install policy for the device. Showing an app satisfies this predicate is usually the goal of evaluating AppPAL.

`App isAnApp, Policy isAPolicy...` Expresses a simple typing relations. AppPAL, and SecPAL, have a safety condition that all variable mentioned in the fact, are also mentioned in the condition body (as is the case for Datalog and other logic languages). These typing relations allow an appeal to ground facts to be made so the safety condition is satisfied. This typically is necessary when an assertion has a constraint but no body conditions: we add the trivial typing statement to the body to ensure the assertion is accepted by AppPAL.

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

`Policy isMetBy(App)` Policies let you split app behaviour into sub-policies. For example in Section 3 we showed how Emma's installation policy could be written using three `isMetBy` statements. Splitting policies allows greater control of how each one is checked. We can delegate checking a policy to an expert using the *can-say* statement. We can specify more detailed checks using the *can-act-as* statements.

`Policy shownIsMetBy(Evidence, App)` A variant of the `isMetBy` statement that allows some evidence (a proof) to be given that shows the policy is met.

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.

Consider the *no-tracking-policy* from earlier. 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 [17]). 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 [13]) 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.

AppPAL also helps attribute blame when things go wrong. By modelling the trust relationships in systems we can work out precisely where mistakes were made. A recent example demonstrating this is CVE-2015-2077. Lenovo was found to be shipping laptops with the *Superfish* malware pre-installed. Lenovo had also installed an AV package on their laptops; this AV did recognise Superfish as malware. Unfortunately they had configured it to ignore Superfish. The malware was an ad framework designed to show users products similar to those they viewed on the web. Unfortunately it did so by man-in-the-middling all SSL traffic using a shared private key.

From a user's perspective there was a delegation of trust to Lenovo to detect malware. Lenovo then delegated further to McAfee to supply the antivirus checking; but left an exception that they would allow Superfish. A user might wonder where the breach of trust occurred and who is to blame? Is it the AV failing to spot the malware? Has someone else configured the software incorrectly? How should they fix the problem?

If we write this in the AppPAL policy language the cause of the breach of trust becomes apparent; as does the fix.

```
1 "user" says "lenovo" can-say inf File isSafe.
2 "lenovo" says "mcafee" can-say inf File isSafe.
3 "lenovo" says "C:\System\superfish" isSafe.
```

The fault lies with line 3. Lenovo has caused the problem (they *said* it was safe). The fix is to revoke this statement, or to revoke line 1 and find a different AV supplier.

4.2 Evaluation

To evaluate AppPAL we implement the SecPAL evaluation rules shown in Figure 4. We do not use the DatalogC [28] based translation and evaluation algorithm suggested by Becker et al.. Rather we implement the rules directly in Java. Pseudo-code is given in Figure 5. 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.

In SecPAL there is no type. All entities are either constants (implemented as strings) or variables. There is no differentiation made between a *voiced* constant (who says assertions), and a *subject* (the subject of a verb-phrase, or predicate argument). In AppPAL the subjects are usually apps or policies. Apps do not utter facts; so there

$$\frac{AC, D \models \texttt{A says B can-act-as C} \quad AC, D \models \texttt{A says C verbphrase}}{AC, D \models \texttt{A says B verbphrase}}\ \text{can-act-as}$$

$$\frac{AC, \infty \models \texttt{A say B can-say } D \texttt{ fact} \quad AC, D \models \texttt{B says fact}}{AC, \infty \models \texttt{A says fact}}\ \text{can-say}$$

$$\frac{\begin{array}{c}(\texttt{A says fact if fact}_1\texttt{, ..., fact}_k\texttt{ where c}) \in AC \\ \forall i \in 1 \cdots k.\ AC, D \models \texttt{A says fact}_i\theta\end{array} \quad \models \texttt{c}\theta \quad \mathit{vars}(\texttt{fact}\theta) = \emptyset}{AC, D \models \texttt{A says fact}\theta}\ \text{cond}$$

Figure 4: AppPAL's evaluation rules.

```
# ac, d |-? q (using previous-results table rt)
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)

# Try can-say and can-act-as
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
```

```
# Try cond rule
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

# Evaluate the antecedents
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
```

Figure 5: Partial-pseudocode for AppPAL evaluation.

is no need to check them as potential speakers of *can-say* statements. Similarly if a constant never appears as a subject of a statement, as most speakers do not, there is no need to check it as a possible subject for a *can-act-as* statement. We pre-process the assertion context to find *voiced* and *subject* constants. This means we need to search less constants when attempting to replace a variable, speeding evaluation.

4.3 Policy Examples

When writing policies some schemes often come up. We give examples of policies and queries and show how they can be implemented in AppPAL.

Policies for home and work In Section 1 we said Alice might like to have separate rules for home and work. Emma may insist that her policies come into effect whenever Alice is at work, but is okay with Alice breaking them after 5 pm or when she isn't at work. To implement this we add rules that use constraints to decide whether an app should be run. Alice already has the Emma's work policy as the default. Now she just needs to add an exception to the rule.

```
"alice" says App isRunnable
  if "home-policy" isMetBy(App)
  where TimeOfDay() > 17.00.

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

App white-listing by an IT department Inside Emma's company employees in the IT department may white-list certain apps as runnable. Emma delegates to them to decide if an app is runnable.

```
"emma" says "it-department" can-say 0 App isRunnable.
```

Charlie and Diveena work in the IT department. When they have checked an app they say it is runnable.

```
"emma" says "charlie" can-act-as "it-department".
"emma" says "diveena" can-act-as "it-department".

"charlie" says "com.facebook.katana" isRunnable.
"diveena" says "org.thoughtcrime.securesms" isRunnable.
```

Overpriviliged applications Alice is particularly worried about apps stealing her data. She knows that certain apps need certain permissions. For example a photography app needs access to the camera. She checks each app's permissions carefully for things that seem unusual.

Alice's policy can be implemented by checking the permissions of the app, using constraints.

```
"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.
```

Discovering the *normal* permissions for an app can be done by mining large stores of apps, or apps a user has previously installed. Alternately tools like Stowaway [14] could be integrated to check apps.

Digital Evidence In Subsection 4.1 we described how digital evidence could be used to decide if policies are met. Suppose a digital evidence generating tool, such as *Evicheck*, can be used to check a policy. For example that no audio can be recorded without the users consent [32]. Using this tool someone generates a proof certificate that this property is met. This might be the developer of the app, a third party certification service, or someone else; when we check the proof we will discover if it is valid or not.

```
"alice" says Anyone can-say 0
  "recording-consent" shownIsMetBy(Evidence, App).

"skb" says "recording-consent"
  shownIsMetBy("evicheck://evidence123", App).

"alice" says "recording-consent" isMetBy(App)
  if "recording-consent" shownIsMetBy(Evidence, App)
  where
    evicheckCheckEvidence(Evidence, App) = true.
```

In the example here Alice allows anyone to present her with evidence for her policy. A security knowledge base (SKB) offers some evidence, and Alice accepts it as if it checks out.

5 Related work

Authorization logics have been successfuly used to enforce policies in several other domains. The earliest such logic, PolicyMaker [7], was general, if undecidable. Logics that followed like KeyNote [8] and SPKI/SDSI [10] looked at public key infrastructure. The RT-languages [27, 28, 29] were designed for credential management. Cassandra [4] 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 [33] 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 [22]. It has already been used to describe data usage policies [1] and inside Grid data systems [24]. Other work on SecPAL has added various features such as existential quantification [3] and ultimately becoming the DKAL family of policy languages [20, 21]. Gurevich 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 [11], 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

```
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.
```

Figure 6: Kirin and AppPAL policies for stopping apps monetized by premium rate text messages.

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 6). Using this approach they found vulnerabilities in Android, but were ultimately limited by being restricted to permissions and broadcast events.

Whilst this approach is great for malware, it is less suitable for finding PUS. The behaviours and permissions PUS 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 PUS 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.

Work by Lin et al. looked at modelling users' willingness to grant different kinds of permissions [30]. They found four clusters of different users each with different kinds of policies: conservatives who do not like granting many permissions, advanced users who seem to have more complex policies, unconcerned users who'll comfortably grant anything, and fence-sitters (about half of all users) who seemed not to have varying opinions. This suggests that users are applying policies and do care how apps behave. Lin et al. speculate that the fence-sitters seemed not to care because of *warning fatigue*. This suggests a system like AppPAL that can reduce user interaction with permission warnings might be interesting. Using a policy unconcerned and fence-sitting users could take up more privacy preserving policies without having to change their behavior. We have created an Android app that can check approximations of Lin et al.'s policies against apps on a users device. This gives us a demonstration of the power of AppPAL using known policies[3].

There has been a great amount of work on developing app analysis tools for Android. Tools such as Stowaway [14] detect overprivileged apps. TaintDroid [12] and FlowDroid [17] can do taint and control flow analysis; sometimes even between app components. Other tools like QUIRE [9] can find privilege escalation attacks between entire apps. ScanDAL [26] and SCanDroid [18] help detect privacy leaks. Appscopy [16] searches for specific kinds of malware. Tools like DroidRanger [37] scan app markets for malicious apps. Many others exist checking and certifying other aspects of app behaviour.

[3]https://github.com/bogwonch/apppal-checker

6 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[4] 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[5]. 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 should also attempt to learn policies from existing users behavior. Given app usage data, from a project like Carat [31], 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.

[4]Called *Android M*.

[5]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/`

References

[1] B Aziz, A Arenas, and M Wilson. SecPAL4DSA. *Cloud Computing and Intelligence Systems*, 2011.

[2] M Backes, S Gerling, C Hammer, and M Maffei. AppGuard–Enforcing User Requirements on Android Apps. *Tools and Algorithms for the Construction and Analysis of Systems*, 2013.

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

[4] M Y Becker and P Sewell. Cassandra: flexible trust management, applied to electronic health records. *Computer Security Foundations*, pages 139–154, 2004.

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

[6] M Y Becker, A Malkis, and L Bussard. A framework for privacy preferences and data-handling policies. Technical report, Microsoft Research, 2009.

[7] M Blaze, J Feigenbaum, and J Lacy. Decentralized trust management. *Security and Privacy*, 1996.

[8] M Blaze, J Feigenbaum, and Angelos D Keromytis. KeyNote: Trust Management for Public-Key Infrastructures. *International Workshop on Security Protocols*, 1550 (Chapter 9):59–63, January 1999.

[9] S Bugiel, L Davi, and A Dmitrienko. Towards taming privilege-escalation attacks on Android. *Network and Distributed System Security Symposium*, 2012.

[10] C Ellison, B Frantz, B Lainpson, R Rivest, and B Thomas. *RFC 2693: SPKI certificate theory.* The Internet Society, 1999.

[11] W Enck, M Ongtang, and P McDaniel. On lightweight mobile phone application certification. *Computer and Communications Security*, pages 235–245, November 2009.

[12] W Enck, P Gilbert, B G Chun, L P Cox, and J Jung. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *Operating Systems Design and Implementation*, 2010.

[13] S Fahl, M Harbach, T Muders, L Baumgärtner, B Freisleben, and M Smith. Why Eve and Mallory Love Android. *ASIA Computer and Communications Security*, pages 50–61, October 2012.

[14] A P Felt, E Chin, S Hanna, D Song, and D Wagner. Android permissions demystified. *Computer and Communications Security*, pages 627–638, October 2011.

[15] A P Felt, E Ha, S Egelman, A Haney, E Chin, and D Wagner. Android permissions: user attention, comprehension, and behavior. *Symposium On Usable Privacy and Security*, page 3, July 2012.

[16] Y Feng, S Anand, I Dillig, and A Aiken. Apposcopy: semantics-based detection of Android malware through static analysis. In *Foundations of Software Engineering*, pages 576–587, New York, New York, USA, November 2014. ACM Request Permissions.

[17] C Fritz, S Arzt, and S Rasthofer. Highly precise taint analysis for android applications. Technical report, Technische Univerität Darmstadt, 2013.

[18] A P Fuchs, A Chaudhuri, and J S Foster. SCanDroid: Automated security certification of Android applications. *USENIX Security Symposium*, 2009.

[19] M I Gordon, D Kim, J Perkins, L Gilham, and N Nguyen. Information-Flow Analysis of Android Applications in DroidSafe. *Network and Distributed System Security Symposium*, 2015.

[20] Y Gurevich and I Neeman. DKAL: Distributed-Knowledge Authorization Language. *Computer Security Foundations*, pages 149–162, 2008.

[21] Y Gurevich and I Neeman. DKAL 2. Technical Report MSR-TR-2009-11, Microsoft Research, February 2009.

[22] J Hallett and D Aspinall. Towards an authorization framework for app security checking. In *ESSoS Doctoral Symposium.* University of Edinburgh, February 2014.

[23] P Hornyack, S Han, J Jung, and S Schechter. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Computer and Communications Security*, 2011.

[24] M Humphrey, S-M Park, J Feng, N Beekwilder, G Wasson, J Hogg, B LaMacchia, and B Dillaway. Fine-Grained Access Control for GridFTP using SecPAL . *Grid Computing*, 2007.

[25] J Jeon, K K Micinski, J A Vaughan, A Fogel, N Reddy, J S Foster, and T Millstein. Dr. Android and Mr. Hide: fine-grained permissions in android applications. *Security and Privacy in Smartphones and Mobile Devices*, pages 3–14, October 2012.

[26] J Kim, Y Yoon, K Yi, J Shin, and S Center. ScanDal: Static analyzer for detecting privacy leaks in android applications. *... on Security and Privacy*, 2012.

[27] N Li and J C Mitchell. Design of a role-based trust-management framework. *Security and Privacy*, 2002.

[28] N Li and J C Mitchell. Datalog With Constraints. *Practical Aspects of Declarative Languages*, 2562(Chapter 6):58–73, January 2003.

[29] N Li, W H Winsborough, and J C Mitchell. Distributed credential chain discovery in trust management. *Journal of computer security*, 2003.

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

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

[32] N Seghir. A Brief Tutorial for Using EviCheck, October 2014. URL `http://groups.inf.ed.ac.uk/security/appguarden/tools/EviCheck/Tutorial.pdf`.

[33] S Smalley and R Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. *Network & Distributed System Security*, 2013.

[34] V Svajcer and S McDonald. Classifying PUAs in the Mobile Environment. *sophos.com*, October 2013.

[35] C Thompson, M Johnson, S Egelman, D Wagner, and Jennifer King. When it's better to ask forgiveness than get permission. In *the Ninth Symposium*, page 1, New York, New York, USA, 2013. ACM Press.

[36] H T T Truong, E Lagerspetz, P Nurmi, A J Oliner, S Tarkoma, N Asokan, and S Bhattacharya. The Company You Keep. *World Wide Web*, pages 39–50, April 2014.

[37] Y Zhou, Z Wang, W Zhou, and X Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. *Network & Distributed System Security*, 2012.