

# Authorization Logic for Mobile Ecosystems

## Second Year Report

Joseph Hallett

October 13, 2015

This report describes the second year work of my PhD. I give a brief overview of app installation policies and the AppPAL authorisation logic; before describing the work completed this year. I review topics suggested in the thesis proposal; and describe my work implementing AppPAL, exploring store and user policies and looking at protocols for app distribution. I describe what I expect to work on in the final year of my studies; and finally, I give a table of contents and plan for my producing my thesis in the remaining time available to me.

## 1. Introduction

Mobile devices are different to other computers. They have access to greater personal data than a PC; pictures and photos as well as conversations, locations, and address books. They mostly run software sold through stores under the manufacturer's control. People take them to work and on holiday; into their homes and out to the towns and countryside. These devices communicate through wireless almost exclusively: working with cars via bluetooth, the internet via 3G, speakers TVs and other internet-of-things devices through wifi. These differences create a different ecosystem for them to operate in: one that we aim to model and control using authorization logic.

Policies describe the precise conditions for an action to be acceptable to a principal (a user, store or other entity that has to make decisions). Authorization logics are designed to express policies which decide whether a principal, for example a computer user, can perform an action. They have been used to implement access control systems to decide whether users can read or write to files. I believe they can also help users decide which apps they want to use.

The interaction between stores, users, apps and phones gives rise to trust relationships between each of the different entities. A trust relationship is where a user trusts a store to provide them with accurate information about apps; or where a store trusts a developer to accurately describe their apps. Users trust the stores to sell them good, malware-free apps. Some stores will sell any apps, but others are more picky relying on audits and

tools to find the higher quality apps to sell. In the stores users are told about the apps they might want to buy (Figure 1): who developed the app, whether they were a *top developer*, what kind of app it is, age ratings, functionality the app may have, aggregated review scores, descriptions and bugs from both anonymous and not so anonymous users, update times, the price of in-app-purchases and the permissions it will request. How do the device owners know which information to trust? Where does the store get this information from in the first place?

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. Users and companies need to find the apps which suit how they want to use their devices. This can be difficult as it isn't obvious how apps use the data they have access to. There is a large amount of choice. Apps are picked for a variety of reasons: ordering in the store [12], reviews and privacy concerns [9], and security rules from employers. These policies could be written down and used to guide device owners when they are making a decision that goes against them.

For example, one user may wish to only install games that have favourable reviews on their favourite gaming site, and which don't have in-app purchases. For this user searching a store might be frustrating: they need to work out the permissions of the app, cross reference the apps between the store and the review site. Using an authorization logic (like AppPAL) we could express the preferences as a policy and by enforcing it only show the user the apps they wanted in the store, or to warn them when they are looking at an app which doesn't meet their policy. The policy would need to delegate to the review site to decide if the app is good, and to google to decide the category of apps. We might also need to examine the app itself to decide whether it can make in-app purchases<sup>1</sup>.

```
'user' says App isInstallable
  if App hasCategory('game'),
    App isGood
  where hasPermission(App, "IAP") = false.

'user' says 'play-store' can-say 0 App hasCategory(Category).
'user' says 'review-site' can-say inf App isGood.


'review-site' says App isGood
  where reviewSiteScore(App) > 7.
```

If an employer uses a bring your own device (BYOD) scheme in their company they trust users to follow company IT policies when using their phones at work. Using tools like Google's *Apps Device Policy*<sup>2</sup> the employer can restrict the use of specific apps. This is an unsubtle form of control. We want to block apps by describing the characteristics of the apps we want or do not want, a process called *blacklisting*. Blacklisting requires constant updating to keep the list up-to-date. Blacklisting doesn't allow for rich policies

---


<sup>1</sup>On Android the ability to make in-app purchase is controlled through the permissions system.

<sup>2</sup><https://play.google.com/store/apps/details?id=com.google.android.apps.enterprise.dmagent>





# Angry Birds


Rovio Entertainment Ltd. Arcade


 PEGI 3

Offers in-app purchases

 You don't have any devices

 Top Developer

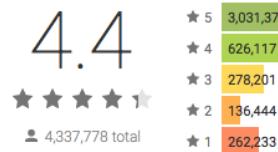
 4,337,778


 Add to Wishlist


Install


## REVIEWS


[Write a Review](#)




**A Google User** ★★★★★  
*Ads! Can't use any power ups, can hit the bolt icon but it just highlights as the ads are in the*


**Buck Apps** ★★★★★  
*One problem: LAG I had this game right when it came out on an android phone and it*


**Katie Hall** ★★★★★  
*Angry Birds I think that all you guys out there reading my comment should defiantly get it*


**Justin F.** ★★★★★  
*A mobile classic Wonderfully drawn art and character designs, a lovely variety of level*



## ADDITIONAL INFORMATION

**Updated**  
August 17, 2015

**Installs**  
100,000,000 - 500,000,000

**Requires Android**  
2.3 and up

**Content Rating**  
PEGI 3  
[Learn more](#)

**In-app Products**  
£0.62 - £25.90 per item

**Permissions**  
[View details](#)

**Report**  
[Flag as inappropriate](#)

**Offered By**  
Rovio Entertainment Ltd.

**Developer**  
[Visit website](#)  
[Email contact@rovio.com](mailto:contact@rovio.com)  
[Privacy Policy](#)  
 Rovio Entertainment Ltd.  
 Keilaranta 17,  
 FI-02150 ESPOO  
 FINLAND

Figure 1: Information presented to the user in the Play store.

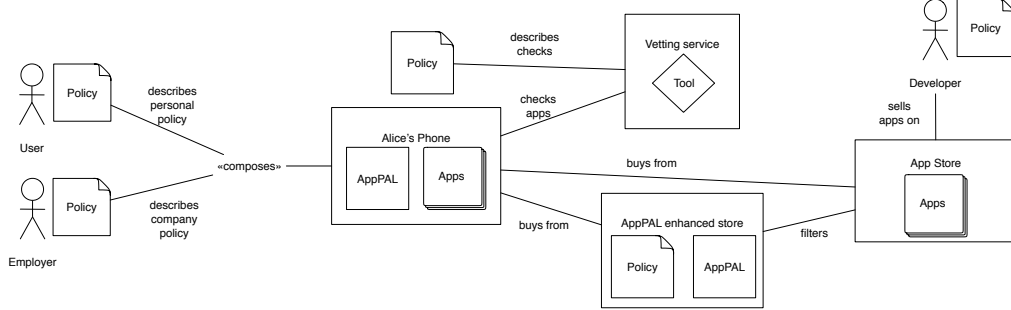


Figure 2: The mobile ecosystem surrounding a phone used for BYOD and the trust relationships surrounding it.

using information from third parties or on the basis of metadata about the app. There is no scope for making any kind of decision other than to allow an app or not. For IT departments in these companies these are the only tools to enforce their policies. For more advanced policies, such as not leaking company documents, compliance is left to employees.

Policies exist for mobile ecosystems (such as the one shown in Figure 2); but there aren't ways of precisely modelling or enforcing them. Authorization logic could be used to describe these policies and enforce them automatically. Logics can describe the trust relationships and policies surrounding Android formally. This lets us make precise statements about the security of these systems. They can be used to make comparisons between different security models, such as those in iOS and Android. It allows comparisons between user's privacy policy and their actual behaviour.

By capturing these patterns as policies they can be enforced automatically. By checking the policies we can enforce them at run-time; or warn when making decisions that go against them. This reduces the burden on users to decide which apps they want. Security-savvy users may design policies themselves: these could be shared with others or used in organisation-wide curated app stores.

This thesis research will show how authorization logic can be used to make security decisions in mobile devices. Currently, security decisions must be made manually by smart phone users. By automating these choices we believe users can avoid having to make security decisions; improving their overall security.

### 1.1. AppPAL

AppPAL is an instantiation of Becker et al.'s SecPAL [3] I have developed to model and enforce app installation policies. Using AppPAL we can describe policies with trust

$$\begin{aligned}
\langle \text{Assertion} \rangle &:= \langle E \rangle \text{ says } \langle \text{Fact} \rangle \\
&\quad (\text{if } (\langle \text{Fact} \rangle, )^+ )?. \\
&\quad (\text{where } \langle \text{Constraint} \rangle )? \\
\langle \text{Fact} \rangle &:= \langle E \rangle (\text{isRunnable} \mid \dots) \\
&\quad \mid \langle E \rangle \text{ can-say } (0 \mid \text{inf}) \langle \text{Fact} \rangle \\
&\quad \mid \langle E \rangle \text{ can-act-as } \langle E \rangle \\
\langle E \rangle &:= \text{Variable} \mid \text{'constant'}
\end{aligned}$$

Figure 3: Simplified grammar of AppPAL.

relationships, and which incorporate constraints. This gives a powerful language to describe user's and company's app policies precisely.

Using AppPAL a principal (someone who utters statements about apps, such as a user or store) can say an app is installable.

```
'user' says 'apk://com.rovio.angrybirds' isInstallable.
```

Or that an app is installable only *if* some conditions are correct and *where* a constraint is true.

```
'user' says App isInstallable
  if 'office-app-policy' isMetBy(App)
  where locationAt('work') = true.
```

There is support for renaming or role assignment.

```
'user' says 'ian' can-act-as 'it-department'.
```

Statements are collected into an *assertion context* which is queried using SecPAL's three evaluation rules [3].

Decisions can be delegated to other principals; with two different forms of the can-say rule where further delegation is allowed (*inf*) or not (*0*). For example as part of a company policy all apps should come with a statement, from an anti-virus, vendor that the app is not malicious. To create this relationship the *can-say* statement is used. A company may trust McAfee to check whether apps are malicious. They do not want McAfee to subcontract the decision making process so they use the *can-say0* form.

```
'company' says 'mcafee' can-say 0 App isMalicious.
```

If McAfee attempts to delegate the decision it will be rejected.

```
'mcafee' says 'f-secure' can-say 0 App isMalicious.
```

With Google, however, they're happy for them to recommend another anti-virus vendor they trust to make the decision for them.

```
'company' says 'google' can-say inf App isMalicious.
```

This time the further delegation would be accepted.

```
'google' says 'f-secure' can-say 0 App isMalicious.
```

## 2. Summary of second year work

In my thesis proposal I stated I would spend this year looking at the role of app stores. Specifically I wanted to understand how users interacted with the stores. This would enable us to better understand their behaviour letting us write policies that accurately described their actions. I said that I would:

- Implement an app store filtered by AppPAL policies.
- Look at how users interact with apps, and the kinds of policies they apply, as well as what happens when their policies change.
- Look at how the policies vary within categories of apps.
- Explore how policies could be composed and joined.
- Explore what happens when policies are attacked, i.e. when apps try and circumvent the policy checks deliberately.

These topics changed as the year went on. I implemented a system for creating app stores based on an AppPAL policy, but I also gained access to data saying what users installed. Rather than working on a system to let users filter apps and then analysing the policies they were using, I tried to look at what apps users pick from stores and the kinds of policies they apply using the information available to them. I started looking at greater detail at the distribution mechanisms and policies within app stores.

Some existing work continued on from the first year: AppPAL was re-implemented in Java to allow it to run on Android. Other work was started from necessity. I wanted a means of systematically running tools over large numbers of apps and collecting the results. So I built a Security Knowledge Base (SKB), a system for automatically collecting security facts about apps, and have used the data collected from it in other parts of the project.

I describe work I have completed this year and show some findings below. My work this year has been exploring the policies surrounding the app stores. To do this I needed a means of storing information about the apps they sell, and a means of checking policies (preferably on a mobile device). From there I could look at the policies specified by the stores themselves, and implicit in the way they operate. Finally I looked at the policies user's want, and compared it to their actual behaviour when using the stores.

### 2.1. An SKB exporting statements to AppPAL.

App stores contain vast numbers of apps. To explore the properties and metadata surrounding the apps we needed a means to systematically analyse them, schedule the analysis tools to run and to store and retrieve results. This would help us organise future work looking at the apps in the stores, and provide a structure to do future research with.

As part of the AppPAL framework I imagined using static analysis results as part of constraints within the language. The SKB was designed to be lightweight and extendable, allowing new tools to be added quickly. Implemented in Ruby the SKB supports running metadata-fetching (information scraped from inside the app or the Play store) and static-analysis tools in parallel over large collections of apps. Adding tools is quick and painless: add a library to Ruby, or a file to a configuration folder implementing a class interface—see Figure 5.

The SKB exports statements to AppPAL, and holds around 50,000 apps. Running the `skb dump` command causes the SKB to emit an AppPAL assertion context that can be used to make queries (Figure 4). The SKB can also emit statements from other principals where appropriate. For example in Figure 4 the SKB emitted statements from the Play store for the apps categorisation and review score. Since the information was obtained from the Play store the SKB issues statements as if the Play store had said them.

Development of the SKB is ongoing work. One possible extension of the SKB could be to store knowledge of vulnerabilities. A device could warn a user if they attempt to install an app with a highly dangerous vulnerability, by checking that no bugs with a high CVSS score are known.

```
'user' says App requiresWarning
  if App hasCVSSScore(X)
    where X > 8.0.
```

The SKB is designed to make fetching and storing these kinds of statements easy. Extending the SKB with additional static analysis tools, and metadata fetching utilities is on-going work. We might also extend the SKB so queries can be made to it through AppPAL constraints. This might allow us to speed up checking some constraints (for example one involving running a static analysis tool) by retrieving the previously computed result from a server instead of recomputing on the device. This would allow us to check policies faster (in some cases). This is desirable as long checking times may make AppPAL unusable.

## 2.2. AppPAL on Android

With a means to collect and store information about apps I wanted to be able to use the information collected to make policy decisions. In the first year I implemented a prototype AppPAL interpreter in Haskell. This was usable to start investigations but could not run on an Android device. Haskell lacks good ARM compilers, and cannot access Android libraries and APIs. I re-implemented it in Java, with a custom evaluation algorithm, and made sure it was portable, running as a library in an app, or Java program. The current implementation contains roughly the same number of lines of code, but runs everywhere and is significantly easier to extend.

Since policy checks may involve inspecting many rules and constraints one may ask whether the checking process will be acceptably fast. Downloading and installing an app takes roughly 30s on an Android phone. If checking a policy delays this even further user's may become annoyed and disable AppPAL.

```

$ skb dump
...
'skb' says 'apk://com.olbg.tipsandroid' can-act-as '34458db1a8d0d1e0bca9bb658cf79872867d97b8' .
'PlayStore' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasReviewScore('4.3').
'PlayStore' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCategory('Sports').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasPermission('android.permission.INTERNET').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasPermission('android.permission.ACCESS_NETWORK_STATE').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasPermission('android.permission.ACCESS_COARSE_LOCATION').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasPermission('android.permission.ACCESS_FINE_LOCATION').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasPermission('android.permission.WRITE_EXTERNAL_STORAGE').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasPermission('android.permission.READ_EXTERNAL_STORAGE').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateVersion('3 (0x2)').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateSerial_Number('2063699619 (0x7b018ea3)').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateSignature_Algorithm('sha256WithRSAEncryption').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateIssuer('0=01bg').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateNot_Before('Aug 13 15:23:16 2013 GMT').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateNot_After('Aug 7 15:23:16 2038 GMT').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateSubject('0=01bg').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificatePublic_Key_Algorithm('rsaEncryption').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificatePublic-Key('(2048 bit)').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateExponent('65537 (0x10001)').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateX509v3_Subject_Key_Identifier('').
'skb' says '34458db1a8d0d1e0bca9bb658cf79872867d97b8' hasCertificateSignature_Algorithm('sha256WithRSAEncryption').
...

```

Figure 4: AppPAL output from the SKB.

```

require 'skb'
require 'zip/zip'

module MetaFetcher
  ##
  # Extracts the certificate used in the APK
  class Certificate < Skb::MetaFetcher
    def execute
      begin
        Zip::ZipFile.open @apk.path do |apk|
          cert = apk.find_entry \
            'META-INF/CERT.RSA'
          unless cert.nil?
            Dir.mktmpdir do |dir|
              cert.extract "#{dir}/CERT.RSA"
              out = `openssl pkcs7
                -in '#{dir}/CERT.RSA'
                -inform DERM
                -print_certs |
                openssl x509 -noout -text`
              @out << out
            end
            return true
          end
          return false
        rescue => _e
          return false
        end
      end
    end
  end
end

require 'skb'

module ResultFetcher
  ##
  # Runs maldroid on an APK
  class Maldroid_HEAD < Skb::ResultFetcher
    ##
    # Timeout after 5 minutes
    def timeout
      300
    end

    def execute
      @out.puts 'maldroid'
      -x
      -f "#{@apk.path}"
      return true
    end
  end
end

```

Figure 5: Fetching plugins for Maldroid and the certificate information for the SKB.



| 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    |

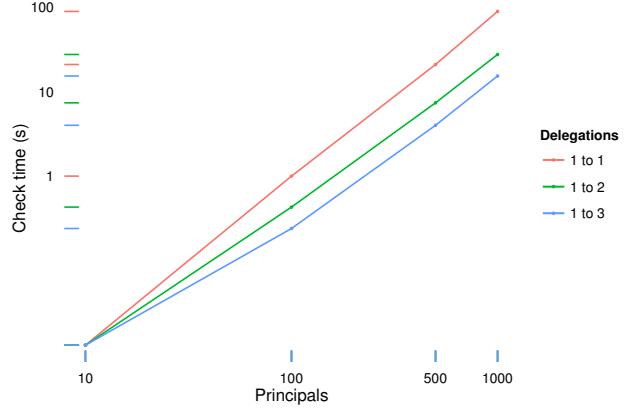


Figure 6: Bench-marking results on a Nexus 4 Android phone.

The policy checking search procedure is at its slowest when having to delegate repeatedly; with the depth of the delegation tree being the biggest factor for slowing the search. Synthetic benchmarks were created to check that the search procedure performed acceptably. Each benchmark consisted of a chain of delegations. The *1 to 1* benchmark consists of a single long chain of delegation. The *1 to 2* benchmark consists of a binary tree of principals delegating to each other. Similarly the *1 to 3* benchmark consists of a tree each principal delegated to three others. These benchmarks are reasonable as they model the worst 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.

On a *Nexus 4* phone checking times are measured in seconds when there are hundreds of delegations, and in minutes when there are thousands (Figure 6). We have only used a few delegations per decision when describing hypothetical user policies. Since the benchmarks show that long chains of delegation can be used, I believe the performance is acceptable, as for most policies the checking time will be under a second.

AppPAL constraints will be the slowest part of checking a policy: since a constraint can execute arbitrary programs or include network requests, the time it takes to run is independent of AppPAL’s checking mechanisms. In practice the only ones I have used are permissions checks. These are very fast as they can call out to the Android package manager, or inspect the APK file directly. Constraints that take a long time to run (such as a static analysis tool) could be used but this may lead to policies designed to limit the number of constraint checks, as they may take too long to check. Current practical policies check small numbers of permissions per app. Since the numbers of constraint checks used in policies are small it is not worth optimising AppPAL’s evaluation algorithm to handle checking large numbers of slow to run constraints (yet).

The library has been integrated into a proof-of-concept policy checking app that scans installed apps against four user privacy policies (Figure 7). This may be extended into something more comprehensive, or into a more polished demo to help promote my

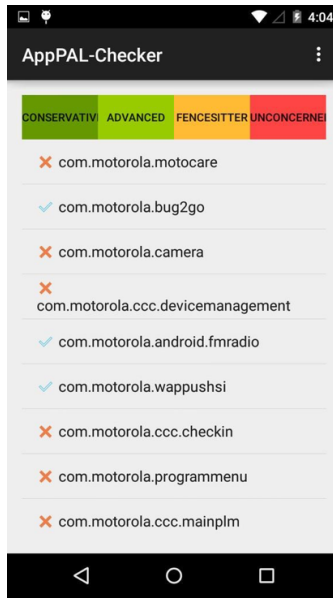


Figure 7: Screen shot from AppPAL checking app.

research.

### 2.3. Explicit policies in stores

With a means of enforcing policies I wanted to look at the policies already being enforced by the marketplaces. One source of these policies is in the user and developer agreements required to use or sell apps in the stores. These policies should give us a good idea about how the stores are used as they are explicit contracts between users, developers and the stores themselves. We might also be able to describe the differences between stores using AppPAL, to help users pick a store that meets their own policies.

There are many Android app stores available. I believed that each of the different stores would have different policies for users and developers, and with different privacy conditions. I read the privacy, developer and user policies for four different app stores (Google Play, Amazon, Aptoide and Yandex) to compare them. In practice I found that the policies tended to be very similar; perhaps even copied in some places. There were some differences when it came to payment processing, and minimum ages to use the store.

Some stores (excepting Google's) kept rights to modify any apps, typically for advertising. This might form the basis of interesting trust relationships. To install an app Android requires its code is signed by a developer. Modifying an app would mean the signature is invalid and so the app would need to be re-signed by the store to be installed. Re-signing is interesting because the trust in the authenticity of the app moves from the developer to the store.

A summary of the different terms and conditions is in Figure 8.

This work could be extended to help users pick an app store that is right for them using a policy. Projects like COAT [5] aim to help users pick a store based on their privacy requirements. We could imagine a similar scheme where AppPAL helps developers decide which stores to submit their apps to, and which stores users should buy from. For example a developer may only agree to sell their app on a store if the store lets them set their own price and they get a 75% cut of the profits.

```
'dev' says 'selling-policy' isMetBy(Store)
  if 'dev' setsPriceIn(Store),
    Cut isCutIn(Store).
  where Cut > 0.75.
```

Figure 8 could be translated into AppPAL and queried to decide which stores can sell the apps; and with this policy only Aptoide would be acceptable.

In Europe and America the choice of which market to use is less interesting. Only Google's Play store (and to a lesser extent Amazon's app store) has enough market share to be compelling. In China, however, where the Play store is banned by the government, there is a greater choice of marketplace and this becomes a more compelling avenue of research.

## 2.4. Distribution mechanisms in app stores.

Whilst the user and developer agreements describe what the apps stores say they will do; to see what they do in practice we need reverse engineering. A user buying an app probably wants it sent to their device. Using an SSL proxy I started looking at the distribution mechanisms, and started reverse engineering their protocols. For simple website stores, such as Opera's mobile store, where apps are distributed without encryption it is trivial to modify downloaded apps on the fly. A man in the middle (MITM) attack using a tool like `mitmproxy` can modify the apps as they are downloaded. In these stores there was no verification that the app downloaded matched the app requested.

Some issues were found in other stores. Amazon's store was the only one to enable certificate pinning. Certificate pinning can prevent SSL sniffing attacks as it requires that the SSL certificate is one known to the app; rather than one that exists in the device's root of trust. The only pinned certificate, however, was for accessing the login server. Once the login token had been issued, however, the traffic between the store and the device could be intercepted and decrypted freely. This isn't a vulnerability as there were other mechanisms in place within the store to prevent app modification: namely cryptographic signatures that came with every app. It does hint, however, that the login system and the store itself have different security guarantees, which may lead to other problems within the store.

Google's Play Store seemed to sometimes drop encryption when downloading APK files. To download an app from the store the user has to follow a protocol, which we describe abstractly in Figure 9. This was inferred by looking at the network traffic and reverse-engineering the store's APK file. After completing the purchase the user presents their proof-of-purchase token to the store along with a description of the app and their own ID (messages 5 & 6). The store then tells them a server to speak to which provides

|                      | Google Play   | Amazon   | Yandex   | Aptoide  |
|----------------------|---|--|--|--|
| User ID              | Name address and billing details.   | Amazon ID.   | None for free apps, payment details for paid.  | Contact details. No verification but agreement not to lie.   |
| Client info taken    | Installation data, device ID, browsing history, cookies. Can opt out.   | Device ID, network info, location, usage data.   | Device ID, SIM number, Device content, System data, browsing history.  | Transaction history. They may share it with developers.  |
| Customer Payments    | Google Wallet and others at Google's discretion.  | Amazon.  | Approved processor by Yandex or store operator.  | Approved processor by Aptoide.   |
| Who is paid?         | Google Commerce.  | Amazon.  | Developer.   | Store owner.   |
| Prices set by        | Developer.  | Amazon.  | Developer (but Yandex may restrict to set values).   | Developer and store owner.   |
| Refunds              | Only for defective or removed content. A refund may be requested for two hours after purchase.  | No.  | Up to 15 minutes after purchase. No for IAP.   | Up to 24 hours after purchase.   |
| Age of use           | At least 13.  | Any age (with consent of guardian). No alcohol related content below 21.   | At least 14.   | A legal age to form a contract with Aptoide.   |
| Update provision     | You agree to receive updates.   | By default.  | Yes for security and bug-fixes.  | Yes agree to receive updates.  |
| Moderation           | No obligation (but they may).   | Publisher obliged to provide info which may be used to give ratings. Amazon will not check these ratings are accurate. | No obligation (but they may).  | No obligation (but they may). Trusted app mark does not indicate moderation.   |
| Acceptable use       | No use as part of a public performance, or for dangerous activities where failure may lead to death.  |  |  | No modification, rental, distribution or derivative works. You may use the software.   |
| Store rights to app  | Marketing and optimising Android.   | Distribution, evaluation, modification, advertising, and creating derivatives for promotion.                           | Advertising.   | Modification and re-selling.   |
| Withdrawal from sale | Immediate.  | 10 days, or 5 days if for copy-write reasons.  | 90 days. A copy will be retained.  | You may.   |
| Developer ID         | Google account and billing details.   | Amazon ID.   | Email, company name, tax-id.   | Email (preferably a Google developer one).   |
| EULA                 | Default offered.  | Only if it doesn't interfere with Amazon's terms.  | Must be provided.  | Default offered  |
| Content restrictions | No alternate stores, sexual, violence, IP infringing, PII publishing, illegal, gambling, malware, self-modifying or system modifying content. No unpredictable network use. | No offensive, pornography, illegal, IP infringing or gambling content.   | No defects. No illegal, disruptive, sexual, IP infringing, PII stealing, alternative stores, or open-source content. | No displaying or linking to illegal, privacy interfering, violent, PII stealing, IP infringing content. Nothing <i>spammy</i> or with unpredictable network use. |
| Payout rates         | 70% of the user's payment.  | 70% list price (minus card fees).  | 70% net-revenue (minus card fees).   | 75% revenue share (minus card fees).   |

Figure 8: Summary of conditions in different stores.

the app to them without further information (the blank message 7 and message 8). This is implemented using a URL redirect, where  $S$  and  $S'$  represent the servers the client talks to. What we observed, however, is that occasionally the redirect was to a server using HTTP rather than HTTPS. This meant that an attacker sniffing the traffic could see which apps were being downloaded and take a copy for themselves. If the sniffing party was an employer, and usage of the app gave away personal information (say a diabetes app or Grindr), then this could be problematic. By replaying message 7 the user (or a third-party who knew the URL) could re-download the app for up to a week later.

Each of the stores I looked at had a slightly different protocol for buying apps and a different means of downloading them onto the device. In an AppPAL-enhanced app store we might imagine apps being supplied with some assertions about their behaviour.

One way to implement this would be for the store, as part of the download, give the user a bundle of AppPAL statements. The user could filter these to find the statements that are useful for deciding any policies they may have. This forces the mobile device to do extra processing which is undesirable if running on batteries. Working out how to send the AppPAL statements in the download protocols, (and with knowledge distribution in general, proposed in Subsection 3.1) seems interesting.

|    |                         |              |  |  |
|----|-------------------------|--------------|--|--|
| 1. | $C \longrightarrow S:$  | $U, C, a_d$  |  |  |
| 2. | $S \longrightarrow C:$  | $a_d, ?$     |  |  |
| 3. | $C \longrightarrow S:$  | $U, !$       |  |  |
| 4. | $S \longrightarrow C:$  | $a_d, \$$    |  |  |
| 5. | $C \longrightarrow S:$  | $U, a_d, \$$ |  |  |
| 6. | $S \longrightarrow C:$  | $S'$         |  |  |
| 7. | $C \longrightarrow S':$ |              |  |  |
| 8. | $S' \longrightarrow C:$ | $a$          |  |  |

| Symbol | Meaning   |
|--------|---|
| $S$    | Store<br>( <code>android.clients.google.com</code> ).                 |
| $C$    | Client the app store app<br>running on the phone.                     |
| $U$    | User, identified by a token.  |
| $a$    | An app.   |
| $a_d$  | a description of the app $a$ .  |
| $?$    | Purchase challenge  |
| $!$    | Purchase challenge response<br>(seemingly derived from $?$ and $U$ ). |
| $\$$   | Download token.   |
| $S'$   | Alternate store URL   |

In message 6,  $S$  redirects  $C$  using a 302-redirect message. By replaying message 7 I found I could re-download the app on a different client. I found that the redirection to the other server sometimes did not use SSL.

Figure 9: Abstraction of the protocol used by Google's Play store to purchase an app.

## 2.5. User app installation behaviour

Having explored the policies and protocols in the stores we also wanted to explore how the stores are used in practice. Users have policies about what data apps should be collecting. Lin et al. [10] identified four different user-privacy policies; however they did not look at whether they enforced them in practice. Using AppPAL I implemented a

simplification of the policies Lin et al. identified. The Carat project [11] collected app installation data from users who installed their energy tracking app. They shared this data with us in an anonymised form where I could see who had installed what; provided I knew the package names of the apps they installed. Using the SKB I de-anonymised 4,300 apps (5%) which accounted for 50% of all app installs. I selected 44,000 users for whom I knew 20 or more app installs.

With the Carat data I measured the extent each user conformed to the privacy policies. I also took a list of known malware and potentially unwanted programs (PUPS) from McAfee to measure the extent of malware installations on Android. I tested whether users who enforce their privacy policies install less malware.

I found that very few users follow a privacy policy all the time [8]. A few users, however, seem to be installing apps meeting a policy 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 users more than 80% of the time. This suggests that while users may have privacy preferences the majority are not attempting to enforce them.

I found 1% of the users had a PUP or malicious app installed. A user is three times more likely to have a PUP installed than malware. Only nine users had both a PUP and malware installed. Users who were complying more than half the time with the conservative or advanced policies complied with the malware or PUP policies fully. This is significant ( $P\text{-value} < 0.05$ ) and suggests that users who pick their apps carefully are less likely to experience malware.

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

### 3. Suggested topics for next year

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

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

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

include bug-fixes so it is in the user’s interests to accept them (despite what they may believe [15]). Implementing methods to weaken security does not seem to be worthwhile. Using AppPAL as a modelling language to precisely describe the trust mechanisms and delegations in an update is interesting. It might make sense to include a comparison of the different update mechanisms in Android and iOS in the thesis.

Policy modification is less interesting. Apps should be rechecked before they can be allowed to run again. If they now break the policy, the user should be prompted to remove the app or make an exception. Results from long-running analysis tools can be stored by the SKB. These could be reused by AppPAL’s constraint mechanisms to avoid re-computation, if no other changes were made.

The app collusion problem is interesting. Whilst it is conceivable that AppPAL policies could describe and prevent attacks, to actually implement the protection in a way that could be used would probably require modifications to the intent system, and potentially binder (Android’s primary IPC mechanism). This probably makes it out of scope of the PhD as we are looking at Android, not a modification of it.

The mechanisms for mechanically composing policies in AppPAL are trivial<sup>3</sup> but what might happen when two principals actively disagree is more interesting. AppPAL doesn’t have support for negation, but it feels right that there should be some way to distinguish between a principal saying an app meets, does not meet, or does not know whether it meets a policy. Work next year should look at the protocols for distributing AppPAL statements.

### 3.1. Knowledge distribution protocol

AppPAL can enforce policies and I have started looking at the kinds of policies users might want to use. When describing AppPAL policies I have thought about delegation relationships and how different principals can be trusted to help make decisions. What isn’t necessarily clear is how these principals get to make these statements and how they are transferred from speaker to speaker.

On a memory constrained device, like a mobile phone, storing a database of all possible AppPAL statements by every speaker is not viable. There are almost 1.5 million apps available on the Google Play store not including other stores or multiple versions of the same app. Storing data on all apps will not work as the file size will be huge. In the papers on SecPAL [3, 2] (on which AppPAL is based) there is little talk about how the knowledge should be distributed. Becket et al. describe how principals should sign their AppPAL statements to identify themselves as having said them, and how an X.509-style public key system could be used to tie keys to principals. This tells us how to check the statements are not forged but does not give the distribution mechanisms. Related languages [4, 1, 6, 7] extended SecPAL features but still didn’t specify the distribution protocol. SecPAL was built to be a distributed language with principals delegating statements. Without a distribution protocols some scenarios can become difficult.

Consider Alice who wants to install an app on her phone. Her installation policy

---

<sup>3</sup>`‘user’ says ‘composed-policy’ isMetBy(App) if ‘pol1’ isMetBy(App), ‘pol2’ isMetBy(App).`

requires confirmation the app is not malware. She knows that McAfee can say (possibly with further delegation) whether an app is malicious or not; but this is a new app and she does not have any prior information about it. She needs to get more information.

One, very simple, approach to distribute the knowledge might be to use a protocol such as:

$$\begin{array}{l} \hline 1. \quad U \longrightarrow S: \quad q, d?, m \\ 2. \quad S \longrightarrow U: \quad r, t, \{r, t\}_{S_{SK}} \\ \hline \end{array}$$

A user ( $U$ ) wishes to find out if a server ( $S$ ) knows any information that will help them answer a query ( $q$ ). To do this they send the server the query along with any metadata they have, for example the app itself, to the server and say whether they'll allow the server to delegate ( $d?$ ). In reply the server sends any information it thinks would be helpful ( $r$ ), plus a timestamp ( $t$ ) and a signature on the response and timestamp so the user is sure they're receiving fresh information. If the user finds they need more information then the protocol will need to be replayed.

This gives a simple protocol for distributing knowledge but it doesn't deal with all the complexities of knowledge distribution. In particular:

- If the app is new McAfee may not know about it either. How should she send it to them for analysis? In the protocol we suggested the app itself could be sent as part of the metadata relating to the query. There may be legal issues surrounding Alice redistributing the app to McAfee, however. If the app is large and Alice has a limited data-allowance she may not want to send the whole app. She could just send a link to the app on an app store, but McAfee may not want to check it if it costs money to download, or the terms of buying the app prevent reverse engineering.
- How do McAfee respond when they can't answer the query? AppPAL does not contain negation but in this scenario it would be helpful to distinguish a statement from McAfee that the app is safe, from one where they know it is definitely malware, from one where they are unsure and wish to err on the side of caution and not make a definite statement.
- What should Alice do while she waits: keep waiting, fail and keep a note to recheck later or fail and never ask again about the app? Should she say how long she is willing to wait so McAfee can distinguish between needing an answer now (Alice is trying to use the app right now) or that the answer isn't urgent (her phone is caching information about apps she hasn't bought yet in case she buys one of them in the future).
- If McAfee wish to delegate the decision they could issue another *can-say* statement. Should McAfee send the public key and any statements from the delegated party to the user, or just the server address and a reference to the public key on a PKI server?



- When should Alice ask McAfee? Before any evaluation would be the easiest time as it would require no change to the evaluation algorithm but during the evaluation might be more appropriate as statements could be imported as needed.

Work this year should look at developing a strategy, and defining a protocol to share and acquire knowledge through AppPAL statements on demand. This would be a worthwhile contribution, as it is novel, and would help extend AppPAL from a SecPAL instantiation to its own language. To do this we aim to complete the following goals:

- Define a protocol where a principal can query another principal for information. The protocol must define precisely what should happen when a further delegation occurs.
- Show that AppPAL (or SecPAL) is still sound and complete when using this protocol as part of its evaluation.
- Integrate the protocol with the AppPAL library and check the performance impact from using it. Explore different strategies for what to do while waiting for answers, and how AppPAL could be configured to switch between them.
- Demonstrate use of the protocol using the SKB as a third-party who can be queried for information.

### 3.2. Case study

Having an authorization logic model and enforce a larger, currently implemented policy shows the limitations and expressivity of a language. It also shows that it is applicable to current compliance problems and is more than just a toy. AppPAL policies I have used so far have been synthetic. This has been fine for testing and demonstrating the language, but a larger example drawn from a real security policy would help give AppPAL credibility.

A natural source for this policy is a corporate BYOD policy where employers restrict the usage of mobile devices in the workplace. The US National Institute of Standards and Technology (NIST) have published recommendations for mobile device security and BYOD in the workplace [14, 13]. Translating the recommendations into AppPAL and showing their enforcement would make a good case study. The study would focus on any difficulties in translating the policies and the extent the recommendations could be enforced automatically. Specifically we would aim to:

- Present an AppPAL policy derived from the guidelines in Sections 2–3 of NIST-SP-800-46 (which gives guidelines for telework and remote access mobile security) and Section 2 of NIST-SP-800-124 (guidelines for mobile security).
- Show where the strengths and limits of AppPAL lie when enforcing the guidelines.
- It is reasonable to expect that we cannot automatically enforce every aspect of the guidelines as they are high level and written in natural language. Some aspects

of the guidelines will need a human to certify that it is met. Using AppPAL we can show where a human will be needed and measure the extend of automatic enforcement.

## 4. Plan for thesis submission

*Hypothesis.* Automatic tools and policy languages would provide a better means of enforcement for mobile device policies than existing mechanisms which rely on manual inspection.

I have 18 months remaining, with my funding running out in March 2017. A Gantt chart showing my plans for the time remaining is included in Appendix A. The next year will be primarily spent completing the projects described in Section 3; and continuing any other work that seems interesting as opportunities arise. Once my review is completed I plan to spend at least one day a week writing up work for my thesis; focussing on the literature review, and completed investigations. From October 2016 I plan to spend the majority of my time writing (as suggested in my thesis proposal). From January 2017 I plan to be predominantly focusing on editing. I believe this is a reasonable plan based on my writing speed, and the time I typically spend editing documents.

Below is a proposed table of contents for the thesis, with content suggestions for each chapter.

1. Introduction
2. Mobile ecosystems

### **App stores and mobile app deployment**

- Introduce app stores and current development practices.
- Show the differences between different market places and the platforms (iOS vs Android).

### **Android Security Model**

- Give overview of Android security architecture.
- Describe how apps work and interact with the system from a technical perspective.
- Introduce the android permissions model and how different apps can access functionality provided by the platform.
- Show how app signing works.

### **Android policies**

- Introduce the need for policies and the motivation for the PhD.
- Show how users do not seem to be following their privacy policies.
- Explain how some companies have mobile device policies, and how users are frustrated with data leaks.

- Show existing tools, such as Kirin, and explain why they're not good enough.
3. AppPAL Implementation
    - The need for a policy language**
      - Introduce scenarios where AppPAL can be used to enforce a policy
      - Show trust relationships between different principals (a user at work).
      - Start to introduce the language.
    - Design and implementation of AppPAL**
      - Formally present the language, as an instantiation of SecPAL.
      - Show evaluation algorithm.
  4. AppPAL Experimentation
    - Deployment**
      - Show applications using AppPAL.
      - Stores generated by policy.
      - On device policy checking.
      - Device configuration by policy (if I can get access to Android M features)
    - Distribution**
      - Define protocol for knowledge acquisition.
      - Describe implementation of protocol.
      - Implement the protocol as part of AppPAL's evaluation algorithm.
  5. Evaluation
    - User privacy policies**
      - Show privacy paradox with Android apps.
      - Use Carat (and McAfee?) data as a model of what users have installed.
      - Show power of AppPAL as a query and modelling language.
      - Discuss the policies users do seem to be enforcing in practice.
    - Case study**
      - Present NIST guidelines for mobile device guidelines for managing the security of mobile devices in the enterprise expressed in AppPAL.
      - Show the language is expressive enough to describe enterprise policies.
      - Discuss problems translating the recommendations, and where the guidelines could not be checked without manual intervention.
  6. Related work
  7. Future work

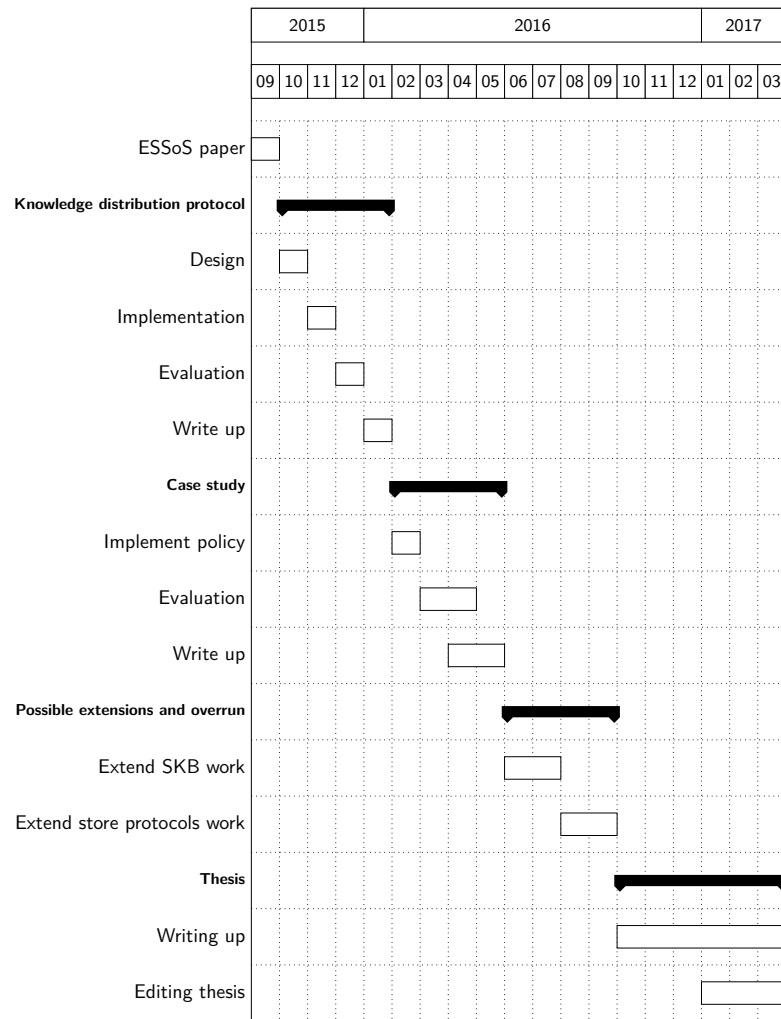
## References

- [1] B Aziz, A Arenas, and M Wilson. SecPAL4DSA. *Cloud Computing and Intelligence Systems*, 2011.
- [2] M Y Becker. Secpal formalization and extensions. Technical report, Microsoft Research, 2009.
- [3] M Y Becker, C Fournet, and A D Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Computer Security Foundations*, 2006.
- [4] M Y Becker, A Malkis, and L Bussard. A framework for privacy preferences and data-handling policies. Technical report, Microsoft Research, 2009.
- [5] C Fernandez-Gago, V Tountopoulos, S Fischer-Hübner, R Alnemr, D Nuñez, J Angulo, T Pulls, and T Koulouris. Tools for Cloud Accountability. *IFIP Privacy and Identity*, 2015.
- [6] Y Gurevich and I Neeman. DKAL: Distributed-Knowledge Authorization Language. *Computer Security Foundations*, pages 149–162, 2008.
- [7] Y Gurevich and I Neeman. DKAL 2. Technical Report MSR-TR-2009-11, Microsoft Research, February 2009.
- [8] J Hallett and D Aspinall. Poster: Using Authorization Logic to Capture User Policies in Mobile Ecosystems. In *Symposium On Usable Privacy and Security*, pages 1–2, July 2015.
- [9] P G Kelley, L F Cranor, and N Sadeh. Privacy as part of the app decision-making process. In *Special Interest Group on Computer-Human Interaction*, pages 3393–3402, New York, New York, USA, April 2013. ACM Request Permissions.
- [10] J Lin, B Liu, N Sadeh, and J I Hong. Modeling Users’ Mobile App Privacy Preferences. *Symposium On Usable Privacy and Security*, 2014.
- [11] A J Oliner, A P Iyer, I Stoica, and E Lagerspetz. Carat: Collaborative energy diagnosis for mobile devices. In *Embedded Network Sensor Systems*, 2013.
- [12] W Prata, A de Moraes, and M Quaresma. User’s demography and expectation regarding search, purchase and evaluation in mobile application store - Work: A Journal of Prevention, Assessment and Rehabilitation - Volume 41, Supplement 1/ 2012 - IOS Press. *Work: A Journal of Prevention*, 2012.
- [13] K Scarfone, P Hoffman, and M Souppaya. NIST Special Publication 800-46: Guide to Enterprise Telework and Remote Access Security, June 2009.
- [14] M Souppaya and K Scarfone. NIST Special Publication 800-124: Guidelines for Managing the Security of Mobile Devices in the Enterprise, June 2013.

- [15] K Vania, E Rader, and R Wash. Betrayed by updates: how negative experiences affect future security. In *Computer Human Interaction*, pages 2671–2674, New York, New York, USA, April 2014. ACM.



## A. Gantt chart



## **B. Paper for ESSoS 2016**

The attached paper is a draft being prepared for the ESSoS<sup>4</sup> symposium. The paper will introduce the AppPAL language and its implementation. I give sanity checking benchmarks to show its performance is not unreasonable. Finally, I demonstrate AppPAL by presenting our work using AppPAL to filter user's privacy policies from our SOUPS poster, as part of a full paper.

---

<sup>4</sup>Engineering Secure Software and Systems



# AppPAL for Android

## Capturing and Checking Mobile App Policies

Joseph Hallett and David Aspinall

School of Informatics, University of Edinburgh

**Abstract.** It can be difficult to find mobile apps that respect your security and privacy. Businesses rely on employees enforcing company mobile device policies 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 [19] and most users do not understand how permissions relate to the capabilities of an app [31]. To address these problems, we present AppPAL: a machine-readable policy language for Android that describes precisely when apps are acceptable. AppPAL goes beyond existing policy enforcement tools, like Kirin [15], adding delegation relationships to allow a variety of authorities to contribute to a decision. AppPAL also acts as a “glue”, allowing connection to a variety of local constraint checkers (e.g., static analysis tools, packager manager checks) to combine their results. As well as introducing AppPAL and some examples, we apply it to explore whether real users follow certain intended policies in practice, finding privacy preferences and actual behaviour are not always aligned, in the absence of a rigorous enforcement mechanism.

## 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 from tools to check apps for problems and issues like SSL misconfigurations [17]. 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, screenshots and 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 [19,42]. 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 [43,41]. 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. 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<sup>1</sup> 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. AppPAL 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 [16] 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:

|  |  |
|--|--|
| <pre>"alice" says App isRunnable   if "home-policy" isMetBy(App)   where at("work") = False.</pre> | <pre>"alice" says App isRunnable   if "work-policy" isMetBy(App)   where beforeHourOfDay("17") = True.</pre> |
|--|--|

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. For example it would be okay allowing a photography app access to the camera, but not to location data if the user doesn’t want their photos geotagged.

```
"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.
```

---

<sup>1</sup> 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.

There has been a great amount of work on developing app analysis tools for Android. Tools such as Stowaway [18] detect over-privileged apps. TaintDroid [14] and FlowDroid [21] 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 [32] and SCanDroid [22] help detect privacy leaks. Appscopy [20] searches for specific kinds of malware. Tools like DroidRanger [45] scan app markets for malicious apps. Various tools such as AppGuard [2], Dr. Android & Mr. Hide [30] or AppFence [28] can control the permissions or data an app can get. MalloDroid [17] looks for apps configured to use SSL incorrectly (for instance by not verifying hostnames or certificates). 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 owners are trying to enforce. This avoids creating tools with hard-coded fixed policies. For example a store might not want to sell apps with SSL errors or malicious apps as determined by their anti-virus. Using AppPAL we can combine tools for checking apps to implement the store’s policies.

```
"play-store" says App isSellable
  if App isAnApp
    where mallodroidCheck(App) = True,
      mcafeeAVCheck(App) = True.
```

## 2 Enforcing A Policy At Work

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

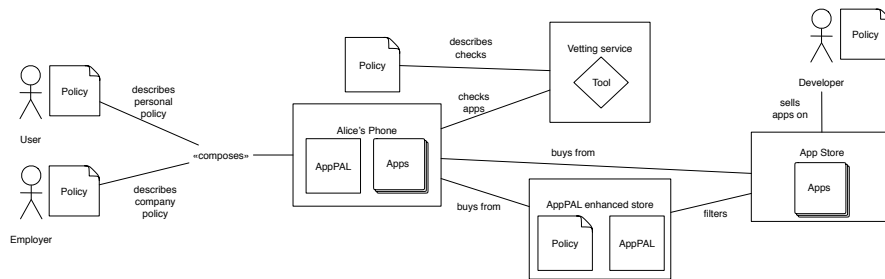
- Alice shouldn’t run any apps that can track her movements. Alice’s workplace is at a secret location and it mustn’t be leaked.
- Apps should come from a reputable source, such as the Google Play Store.
- Emma uses an anti-virus (AV) program by McAfee. It should check all apps before they’re installed.

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* [24] could configure Alice’s device to disallow apps from outside the Google Play Store and let Emma set the permissions granted to each app [37].

We could implement Emma’s policy using existing tools (such as an AV checker, and a taint analysis tool like Flowdroid [21]) but it is a clumsy solution—they are not flexible. Each has to be configured separately to implement only part of the policy. 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 disabled 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 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 isBuyable.
12
13 "emma" says "google-play" can-say
14 App isBuyable.
15
16 "emma" says "anti-virus-policy" isMetBy(App)
17 if App isAnApp
18 where
19   mcAfeeVirusCheck(App) = False.
20
21 "emma" says "no-location-permissions"
22 can-act-as "no-tracking-policy".
23
24 "emma" says
25 "no-location-permissions" isMetBy(App)
26 if App isAnApp
27 where
28   hasPermission(App, "LOCATION")=False.

```

**Fig. 2.** AppPAL policy implementing Emma’s security requirements.

In line 2 Alice and Emma specify whether an `App` (a variable) `isRunnable`; she allows her to delegate the decision further (`can-say inf`). Emma specifies her concerns as policies to be met in line 4: if Emma is convinced all these 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 isBuyable`; with `"google-play"` as the decider of what is buyable or not. Google is not allowed to delegate the decision further, i.e. Google is not allowed to specify Amazon as a supplier of apps as well. Emma specifies the `"anti-virus-policy"` in line 15 using 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 she needs to collect statements to show the app meets the `isRunnable` predicate. Namely:

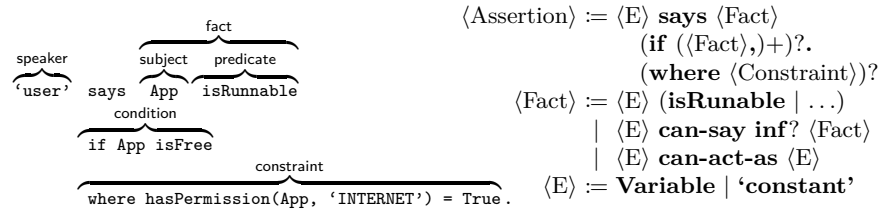
- `"google-play" says "com.facebook.katana" isReputable`. Required to convince Emma the app came from a reputable source.
- `"emma" says "anti-virus-policy" isMetBy("com.facebook.katana")`. She can obtain this by running the AV program on her app.
- `"emma" says "no-locations-permissions" isMetBy("com.facebook.katana")`. 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 second statement AppPAL 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 third statement the AppPAL checker needs to examine 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.

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.



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

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.

| Name                                   | Description                                     |
|--|---|
| App <b>isRunnable</b>                  | Says an app can be run.                         |
| App <b>isInstallable</b>               | Says an app can be installed.                   |
| App <b>isAnApp</b>                     | Tells AppPAL that an app exists.                |
| Policy <b>isMetBy</b> (App)            | Used to split policies into smaller components. |
| <b>hasPermission</b> (App, Permission) | Constraint to check if an app has a permission. |
| <b>beforeHourOfDay</b> (time)          | Constraint used to check the time.              |
| <b>ToolCheck</b> (App, Property)       | Constraint to run an analysis tool on an app.   |

**Table 1.** AppPAL predicates and constraints.

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 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 [21]). 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 (using MalloDroid [17]) 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, as well as facts deduced while checking.

While Becker et al. used a DatalogC based checking algorithm, we have implemented the rules directly in Java as no DatalogC library is currently available for Android. Pseudo-code is shown in Figure 4.

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. When delegating we will also be able to request facts from the delegated party dynamically; though implementing this is future work.

```

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 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.sub a).variables == none
      return checkConditions ac, rt, a, d
  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.

## 4.2 Benchmarks

When AppPAL runs on a mobile phone apps should be checked as 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 the principals. In the *1 to 2* benchmark each principal delegated to 2 others and in the *1 to 3*



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 negligible.

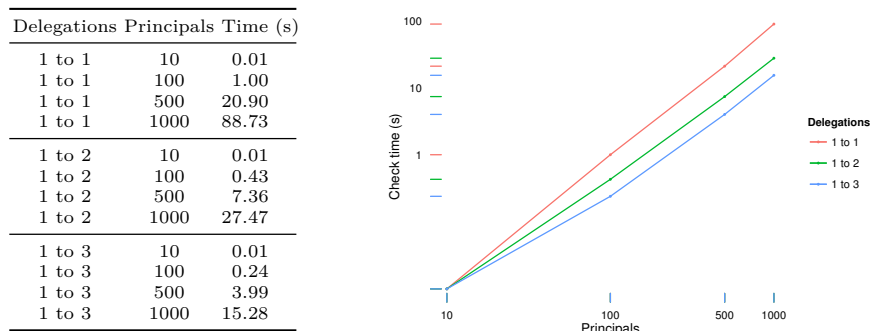


Fig. 5. Benchmarking results on a Nexus 4 Android phone.

## 5 Measuring Policy Compliance

Throughout we have asserted that users 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 [38,40]. In a study of 725 Android users, Lin et al. found four patterns that characterise user privacy preferences for apps [35] demonstrating a refinement of Westin’s privacy segmentation index [33]. Using app installation data from Carat [36,11] we used AppPAL to find the apps satisfying each policy Lin et al. identify and measure the extent that each user was following a policy.

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 by Lin et al. as we do not take into account the reason each app might have been collecting each permission. Using AppPAL we could write more precise rules if we could determine why each permission was requested. Lin et al. used Androguard [12] as well as manual analysis to determine the precise reasons for each permission [35].

|                        | Policy C A F U |   |   |   |   |
|------------------------|----------------|---|---|---|---|
| GET_ACCOUNTS           | X              | X | X | X | X |
| ACCESS_FINE_LOCATION   | X              | X | X |   |   |
| READ_CONTACT           | X              | X | X |   |   |
| READ_PHONE_STATE       | X              | X |   |   |   |
| SEND_SMS               | X              | X |   |   |   |
| ACCESS_COARSE_LOCATION | X              |   |   |   |   |

It is also interesting to discover when people install apps classified as malware. McAfee classify malware into several categories, and provided us with a dataset of apps classified as malware and PUP. 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.

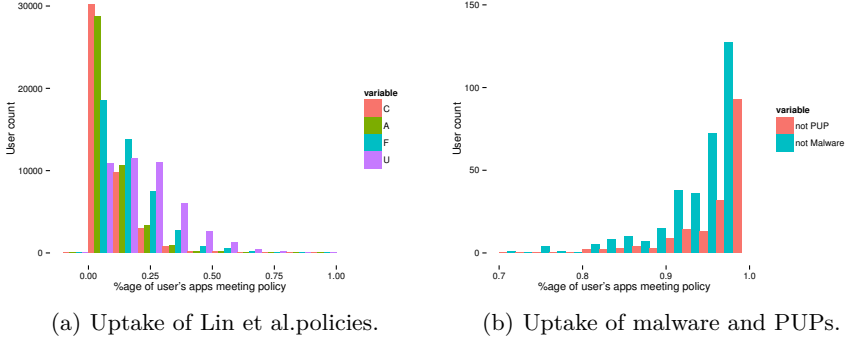
```
"user" says "mcafee" can-say
"malware" isKindOf(App).
"mcafee" says "trojan" can-act-as "malware".
"mcafee" says "pup" can-act-as "malware".
```

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 now want to test how closely user behavior follows policies. Installation data was taken from a partially anonymized<sup>2</sup> database of installed apps captured by Carat [36]. 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 Carat 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.

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

<sup>2</sup> Users are replaced with incrementing numbers, app names are replaced with hashes to protect sensitive names.



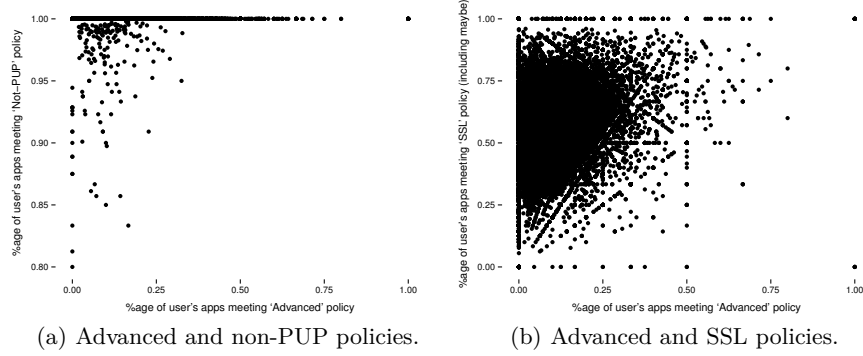
**Fig. 6.** Policy compliance graphs. Each histogram shows the number of users who followed a policy to a certain extent. Users who installed no malware have been omitted from Figure 6(b).

and their behaviour (reminiscent of the *privacy paradox*); assuming the user population studied by Lin et al. behave similarly to data from the Carat study. 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 ad hoc, manual 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.

The *MalloDroid* tool [17] can scan apps for SSL misconfigurations. SSL misconfigurations are dangerous as they can undermine any privacy guarantees that SSL/TLS usage gives. MalloDroid can scan for several misconfigurations, and distinguishes cases where the app is definitely misconfigured from those where there is some doubt. We set up AppPAL to use MalloDroid results as a constraint and measured the percentage of apps each Carat 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, see Figure 7(b). This emphasizes that AppPAL can help enforce complex policies that cannot be checked for without additional tools.

There are some limitations in this investigation: firstly 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



**Fig. 7.** Compliance with the advanced policy and the non-PUP and SSL policies. Each data-point represents a user. In (a) we see that users who followed the Advanced policy more than 50% of the time did not install any malware. In (b) we see that even users who followed the Advanced policy were no better at avoiding apps with SSL problems than any other users.

the policy without us knowing. Secondly 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 [44]; so a user may be more conservative than our analysis suggests. Finally, as mentioned, we have compared a different set of users to the ones Lin et al. looked at. We plan to do a more comprehensive user study in the future that investigates AppPAL in use with different communities.

## 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 [13] looked at public key infrastructure. The RT-languages [34] 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 [39] 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 fit for the mobile ecosystem setting [27]. It has also been used to describe data usage policies [1] and inside Grid data systems [29]. Other work on SecPAL has added various features such as existential quantification [4] and the DKAL family of policy languages [25,26]. 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 [16] 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, e.g. restricting the use of certain apps while at work. We showed how static analysis tools can be integrated into AppPAL to check for complex properties and to act as a glue between different policies.

AppPAL currently runs either an Android app or a Java program. Further work is needed 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. The API is protected, however, and it would require the phone to have a custom firmware. This is undesirable as it would make AppPAL difficult to install for most users, and negate the other security enhancements (such as timely updates and patches) provided by the standard Android system. 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 [23] (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 policies for specific use-cases<sup>3</sup>. 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 user's behavior. Given app usage data, from a project like Carat [36], 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, rigorous, ways for machines to enforce user's and device-owner's rules about how apps should behave. These policies can be enforced more reliably, and with less interaction from person operating the device.

## Acknowledgements

Thanks to Igor Muttik at McAfee, and N. Asokan at Aalto University and the University of Helsinki for discussions and providing us with data used in Section 5.

---

<sup>3</sup> <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* (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* (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* (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. Chia, P.H., Yamamoto, Y., Asokan, N.: Is this App Safe? *World Wide Web* (Apr 2012)
12. Desnos, A.: Androguard. <https://github.com/androguard/androguard>
13. Ellison, C., Frantz, B., Lainpson, B., Rivest, R., Thomas, B.: RFC 2693: SPKI certificate theory. *The Internet Society* (1999)
14. 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)
15. Enck, W., Ongtang, M., McDaniel, P.: Mitigating Android software misuse before it happens. Tech. Rep. NAS-TR-0094-2008 (Sep 2008)
16. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. *Computer and Communications Security* pp. 235–245 (Nov 2009)
17. 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)
18. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. *Computer and Communications Security* pp. 627–638 (Oct 2011)
19. 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)
20. 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)
21. Fritz, C., Arzt, S., Rasthofer, S.: Highly precise taint analysis for android applications. Tech. rep., Technische Universität Darmstadt (2013)

22. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: SCanDroid: Automated security certification of Android applications. *USENIX Security Symposium* (2009)
23. Google: Android Auto. [com.google.android.projection.gearhead](http://com.google.android.projection.gearhead)
24. Google: Google Apps Device Policy. [com.google.android.apps.enterprise.dmagent](http://com.google.android.apps.enterprise.dmagent)
25. Gurevich, Y., Neeman, I.: DKAL: Distributed-Knowledge Authorization Language. *Computer Security Foundations* pp. 149–162 (2008)
26. Gurevich, Y., Neeman, I.: DKAL 2. Tech. Rep. MSR-TR-2009-11, Microsoft Research (Feb 2009)
27. Hallett, J., Aspinall, D.: Towards an authorization framework for app security checking. In: *ESSoS Doctoral Symposium*. University of Edinburgh (Feb 2014)
28. 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)
29. 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)
30. 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)
31. 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)
32. Kim, J., Yoon, Y., Yi, K., Shin, J., S Center: ScanDal: Static analyzer for detecting privacy leaks in android applications. *Mobile Security Technologies* (2012)
33. Krane, D., Light, L., Gravitch, D.: Privacy On and Off the Internet. *Harris Interactive* 18(5), 345–359 (Oct 2002)
34. Li, N., Mitchell, J.C.: Design of a role-based trust-management framework. *Security and Privacy* pp. 114–130 (2002)
35. Lin, J., Liu, B., Sadeh, N., Hong, J.I.: Modeling Users' Mobile App Privacy Preferences. *Symposium On Usable Privacy and Security* (2014)
36. Oliner, A.J., Iyer, A.P., Stoica, I., Lagerspetz, E.: Carat: Collaborative energy diagnosis for mobile devices. In: *Embedded Network Sensor Systems* (2013)
37. Poiesz, B.: Android M Permissions. In: *Google I/O* (2015)
38. Scarfone, K., Hoffman, P., Souppaya, M.: NIST Special Publication 800-46: Guide to Enterprise Telework and Remote Access Security (Jun 2009)
39. Smalley, S., Craig, R.: Security Enhanced (SE) Android: Bringing Flexible MAC to Android. *Network & Distributed System Security* (2013)
40. Souppaya, M., Scarfone, K.: NIST Special Publication 800-124: Guidelines for Managing the Security of Mobile Devices in the Enterprise (Jun 2013)
41. Svajcer, V., McDonald, S.: Classifying PUAs in the Mobile Environment. [sophos.com](http://sophos.com) (Oct 2013)
42. 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)
43. 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)
44. Wei, X., Gomez, L., Neamtiu, I., Faloutsos, M.: Permission evolution in the Android ecosystem. In: *Annual Computer Security Applications Conference*. pp. 31–40. ACM Request Permissions, New York, New York, USA (Dec 2012)
45. 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)