THE UNIVERSITY *of* EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Capturing Mobile Security Policies Precisely

Joseph Hallett

Doctor of Philosophy

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2017

Abstract

The security policies of mobile devices that describe how we should use these devices are often informally specified. Users have preferences for some apps over others. Some users may avoid apps which can access large amounts of their personal data, whilst others may not care. A user is unlikely to write down these policies or describe them using a formal policy language. This is unfortunate as without a formal description of the policy we cannot precisely reason about them. We cannot help users to pick the apps they want if we cannot describe their policies.

Companies have mobile security policies that define how an employee should use smart phone devices and tablet computers from home at work. A company might describe the policy in a natural language document for employees to read and agree to. They might also use some software installed on employee's devices to enforce the company rules. Without a link between the specification of the policy in the natural language document and the implementation of the policy with the tool, understanding how they are related can be hard.

This thesis looks at developing an authorisation logic, called AppPAL, to capture the informal security policies of the mobile ecosystem, which we define as the interactions surrounding the use of mobile devices in a particular setting. This includes the policies of the users, the devices, the app stores, and the environments the users bring the devices into. Whilst earlier work has looked on checking and enforcing policies with low-level controls, this work aims to capture these informal policy's intents and the trust relationships within them separating the policy specification from its enforcement. This allows us to analyse the informal policies precisely, and reason about how they are used.

We show how AppPAL instantiates SecPAL, a policy language designed for access control in distributed environments. We describe AppPAL's implementation as an authorisation logic for mobile ecosystems. We show how we can check AppPAL policies for common errors. Using AppPAL we show that policies describing users privacy preferences do not seem to match the apps users install. We explore the differences between app stores and how to create new ones based on policy. We look at five BYOD policies and discover previously unexamined idioms within them. This suggests aspects of BYOD policies not managed by current BYOD tools.

Acknowledgements

I am grateful to many people for their help in completing my PhD. In particular I would like to thank the following people:

Thank you to my supervisors David Aspinall and Björn Franke for their advice, criticism, help, patience, and always offering to edit my papers. Thank you to Andy Gordon and Martin Hofmann for the helpful discussions early on, introducing me to SecPAL and to using formal logic to model this domain. Thank you to my examiners, Paul Jackson and Charles Morisset for their comments.

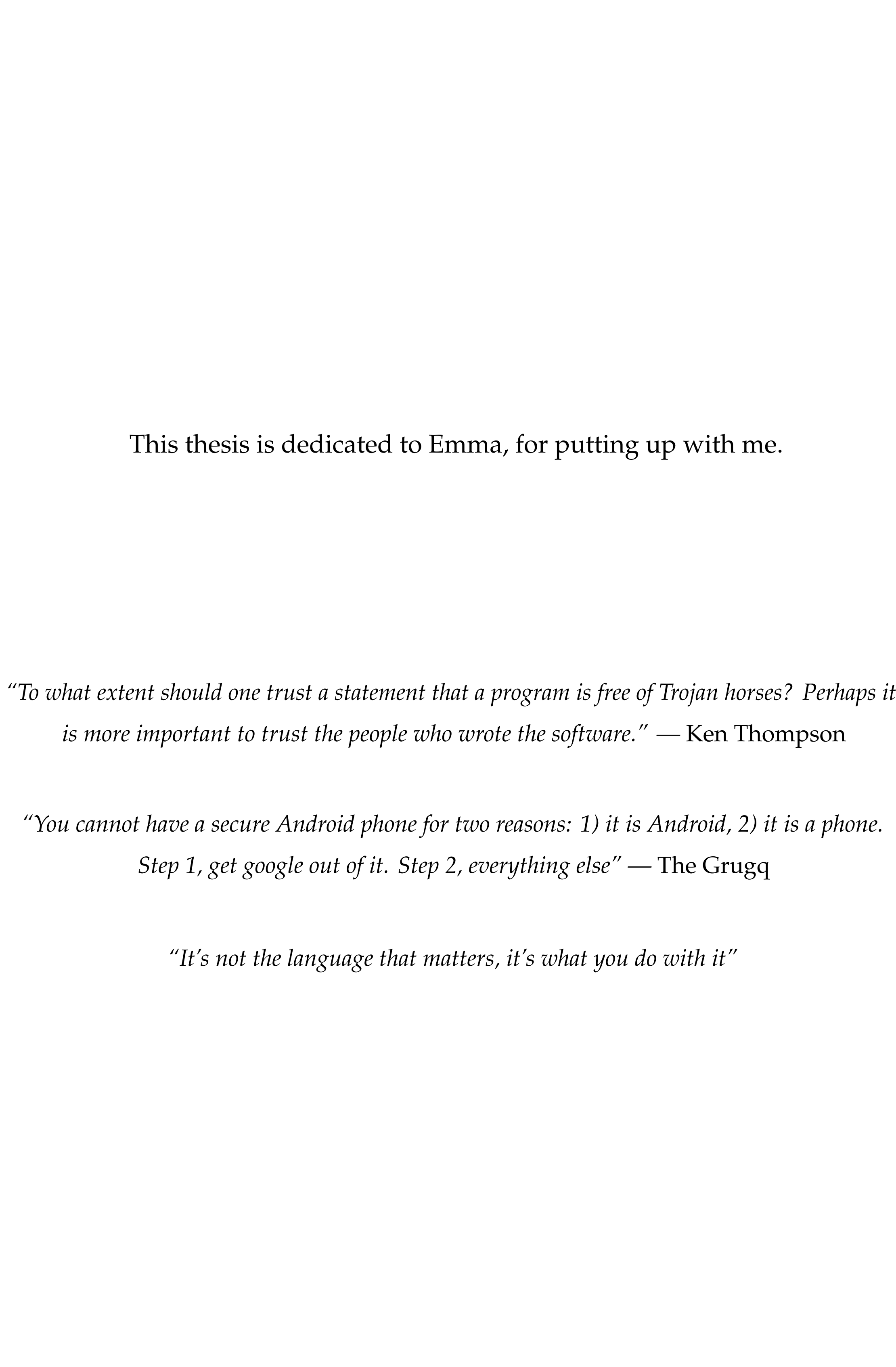
Thank you to Daniel, Marcin, Arthur, Catherine and the rest of IF 5.24 for listening to my rants and helpful discussions over the years. Thank you to Dan Page for his help early in my career, and for suggesting I apply for this job. A second thank you to David Aspinall for helping me move back home to be a dad before I really should have. Thank you to my friends and family for supporting me over the past years, especially Di for her camaraderie in completing this. Also, thanks to EasyJet for roughly 316 flights between Edinburgh and Bristol.

Finally, thank you to Emma and Jim. Thank you Emma for your unwavering support and for letting me work when I should be helping with the baby, and thank you Jim for keeping me awake and writing at all hours.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Joseph Hallett*)



This thesis is dedicated to Emma, for putting up with me.

"To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software." — Ken Thompson

"You cannot have a secure Android phone for two reasons: 1) it is Android, 2) it is a phone. Step 1, get google out of it. Step 2, everything else" — The Grugq

"It's not the language that matters, it's what you do with it"

Table of Contents

1 Introduction **1**

1.1 The Policies of the Mobile Ecosystem 2

1.2 Capturing Mobile Security Policies Precisely 8

1.2.1 Thesis Outline and Publications 10

1.2.2 Research Contributions 12

2 Background and Related Work **14**

2.1 SecPAL 14

2.1.1 Delegation in SecPAL 23

2.2 Policy Languages 27

2.3 Fine-Grained Permission Systems 40

2.4 Moving Forward 42

3 Instantiating and Evaluating SecPAL **43**

3.1 Why SecPAL 44

3.2 Basic Examples of AppPAL 49

3.2.1 AppPAL Policies for App Stores 50

3.2.2 Worked Example of Policy Checking 52

3.3 Instantiating SecPAL for Mobile Ecosystems 55

3.3.1 Predicate Conventions 55

3.3.2 Type Notation 60

3.4 Implementation 63

3.4.1 Evaluation 65

3.4.2 Soundness and Completeness of Decision Procedure . . 69

3.4.3 Benchmarks 73

3.5 Automatic Analysis of AppPAL policies 74

3.5.1 Checking Satisfiability 75

3.5.2 Checking Redundancy . . . 80

4 App Stores and App Preferences **85**

4.1 App Stores . . . 85

4.1.1 Exploring Differences in Terms Between App Stores . . . 87

4.1.2 Why Who Signed the App Matters . . . 94

4.2 Finding the Right Apps . . . 96

4.2.1 Privacy Preferences . . . 97

4.2.2 Measuring Users . . . 99

4.2.3 Privacy Policies and Malware . . . 103

4.2.4 Discussion . . . 105

4.3 An AppPAL Enhanced Store . . . 106

4.3.1 Using GenStore to Build an App Store . . . 107

4.3.2 Current Status . . . 108

5 Applying AppPAL to BYOD Policies **110**

5.1 BYOD in the Workplace . . . 110

5.1.1 Overview of Five BYOD Policies . . . 111

5.1.2 Review of MDM software . . . 113

5.1.3 Related BYOD Work . . . 115

5.2 Modelling BYOD policies . . . 117

5.3 BYOD Idioms in AppPAL . . . 118

5.3.1 Delegation and Roles Within Policies . . . 119

5.3.2 Acknowledgement . . . 122

5.4 Enforcing a BYOD policy with AppPAL . . . 124

6 Future Work **126**

6.1 Probable SecPAL . . . 126

6.1.1 Examples of Probability . . . 127

6.1.2 Guarantees for Probable SecPAL . . . 129

6.2 Patterns with Predicates . . . 131

6.3 AppPAL MDM . . . 133

6.4 Usability Study . . . 134

7 Conclusion **135**

A Translated BYOD Policies **137**
A.1 NHS 137
A.2 SANS 148
A.3 HiMSS 157
A.4 Edinburgh 161
A.5 Sirens 167

B Probabilistic SecPAL Changes and Evaluation **174**
B.1 Evaluating Probability 174
B.2 A Probable Algorithm 5.2 176

Bibliography **178**

List of Figures

1.1 Interactions surrounding the use of mobile devices. 2

2.1 Simple example of a policy scenario. 15
2.2 Structure of a SecPAL assertion. 15
2.3 A policy scenario involving variables. 17
2.4 Simple SecPAL proof. 19
2.5 A more complex SecPAL proof involving variables. 19
2.6 A policy scenario involving constraints. 20
2.7 BNF description of SecPAL. 21
2.8 Derivation rules used to evaluate SecPAL. 21
2.9 SecPAL's proof-theoretic semantics. 22
2.10 SecPAL's assertion safety conditions. 22
2.11 SecPAL's IN/OUT query safety condition. 22
2.12 Example of delegation on a cluster. 24
2.13 Example of unbounded delegation. 25
2.14 Example of delegation with roles. 25
2.15 Example of depth-bounded delegation. 26
2.16 Example of delegation with a loop of trust. 27
2.17 Timeline of the development of different policy languages. . . . 28
2.18 A permissive XACML policy. 36
2.19 Excerpt from XACML delegation example. 38
2.20 The XACML reference architecture. 39

3.1 Entities in the mobile ecosystem 44
3.2 Proof tree output by AppPAL. 56
3.3 Changes to SecPAL's syntax to support types. 60
3.4 De-sugaring from AppPAL types to SecPAL. 61
3.5 Procedure to expand types from AppPAL into SecPAL. 61

3.6 AppPAL's inputs and outputs. 65
3.7 Pseudo-code for evaluating a query. 67
3.8 Pseudo-code for using the cond-rule. 67
3.9 Pseudo-code for using the can-say and can-act-as rules. 68
3.10 Excerpts from the 1 to 1, 1 to 2 and 1 to 3 benchmarks. 73
3.11 Proof graph showing irrelevance. 81
3.12 Proof graph showing unreachability. 82
3.13 A simple policy shown as a graph. 83
3.14 Flattening a more complex policy. 84

4.1 Adware infested and pirated app from Aptoide. 87
4.2 Rooting app found on Aptoide. 87
4.3 AppPAL translations of app store refund rules. 93
4.4 AppPAL translations of app store support rules. 94
4.5 Adoption of Lin et al. policies. 101
4.6 Explanation of adoption charts. 102
4.7 Malware installation numbers in the Carat data set. 104
4.8 Plots of conformance to policies against malware installed. 104
4.9 AppPAL GenStore's architecture. 106
4.10 GenStore database schema. 106
4.11 Output of the GenStore tool. 109

5.1 Policy settings in the MaaS360 MDM tool. 115
5.2 Interactions in a company with BYOD security policies. 124

6.1 VirusTotal results for two Android apps. 128

List of Tables

3.1 Standard prefixes used for AppPAL predicates. 55
3.2 Sets used in AppPAL evaluation. 66
3.3 Benchmarking results on a Nexus 4 Android phone. 74

4.1 Comparison of terms and conditions from five app stores. . . . 92
4.2 Lin et al. policies expressed as sets of permissions. 97
4.3 Lin and Westin privacy groups and their sizes. 98
4.4 Summary of experiment to measure the extent user's follow a policy. 99

5.1 Summary of different MDM capabilities 114
5.2 Summary of method to identify idioms in BYOD policies. 116
5.3 Counts of predicate-types in each policy. 120
5.4 Summary of different authorities in BYOD policies. 120
5.5 Occurrences of predicates common to multiple policies. 121

6.1 Summary of temporal operators from Prior. 131

Chapter 1

Introduction

Mobile devices are ubiquitous, yet the relationships between their users, and the environment they run in is often vague. Users have preferences for the apps they use. Stores have terms for the apps they sell, and for the users who buy them. Companies have policies as to what apps and devices employees can use in the office. The precise trust relationships, however, are often hidden in informal, or natural language, descriptions of the policies.

As the devices have become more powerful, there has been an increasing wish to control the devices. Companies expect devices to follow their corporate policies within their networks and trust users to abide by their policies as well as use tools to enforce some aspects of them. Some users may have reservations about what data an app can get access to and may wish to restrict the app's access. Users may rely on stores to vet the apps they sell, but will likely not know (or necessarily care) precisely how the store checked the app.

These devices exist within the *mobile ecosystem*: which we define as **the interactions surrounding the use of smart phones and tablet computers in a given setting**. Figure 1.1 shows some relationships between devices, their users and their preferences, the stores, companies and all these principal's policies. Users have phones or other mobile devices. The users may own personal or have company provided devices. They may have their own preferred ways of using the device, or they have to follow policies written by their employer. Users download apps, written by developers, from app stores; each store has their own policies and some stores may delegate some aspects of their quality control to external vetting software.

Prior work focused on how systems and tools can check and enforce more

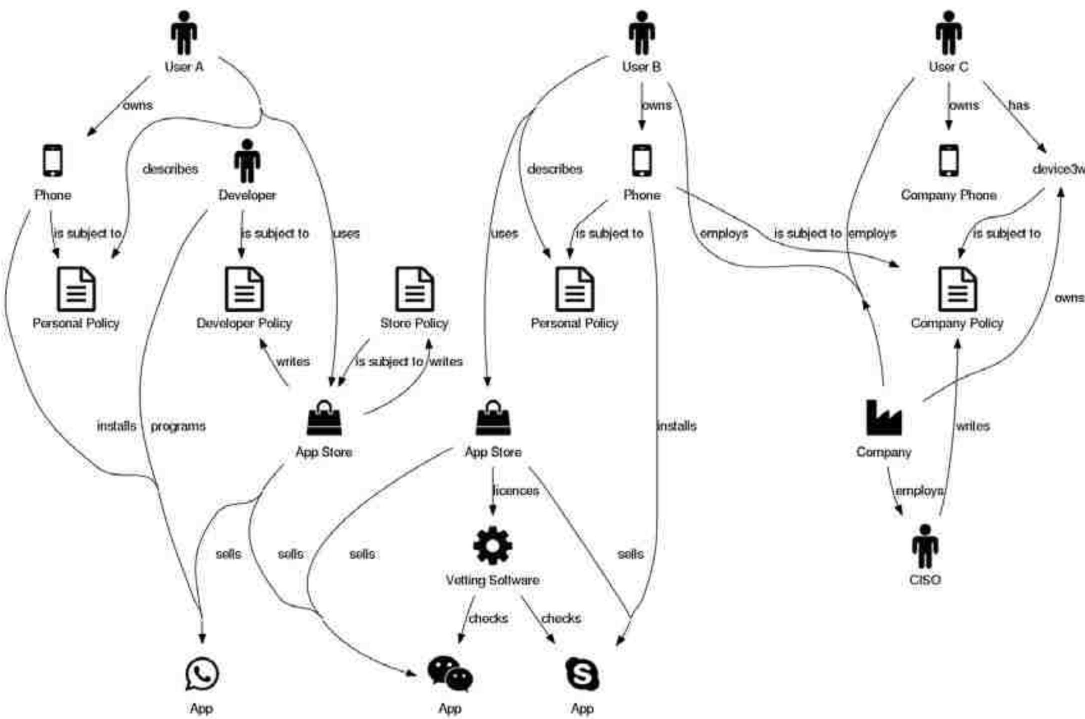


Figure 1.1: Interactions surrounding the use of mobile devices.

sophisticated policies and ever finer controls. This thesis asks a different question: **how can we capture the informal policies and trust relationships surrounding the mobile ecosystem and use formal languages to model and examine them?** We ask how can we tie the top-level goals in the natural language policies and preferences to the tools used to implement them? How can we compare different policies and highlight similarities and differences between them precisely?

1.1 The Policies of the Mobile Ecosystem

As mobile devices are increasingly capable and hold ever-increasing amounts of information, users and businesses need to manage how the devices behave. Employees now bring their mobile devices to work and use them to access company email and documents. In response to this companies might *mandate* that employees follow mobile device policies that describe how the employees should use their devices within the company. These policies vary in terms of formality inside and outside of a company. They may also use Mobile Device Management (MDM) software, tools which allow companies to configure mobile

devices remotely, to enforce the policies. Regulation, such as Health Insurance Portability and Accountability Act (HIPAA), may also affect some companies.

A user may never write their personal privacy preferences in a formal language but they may make decisions guided by them. An example might be a user choosing which apps to install and which to avoid, based on their own *discretion*. They may make decisions based on what their friends have told them, or what a review said about the app.

In this section we will start to introduce by example AppPAL: an authorisation language for the policies of the mobile ecosystem, and is based on SecPAL [23]. We will describe AppPAL in greater detail in Chapter 3 and throughout the thesis, but it is, in essence, an instantiation of the SecPAL system to the mobile domain, along with some minor changes to syntax. We have implemented AppPAL to explore its use and to capture the policies of the mobile ecosystem. Whenever we show an AppPAL snippet in `teletype font` it can be parsed and used as part of a policy[1] with our implementation.

An AppPAL policy consists of many assertions. Each assertion is a statement by a principal either of a fact, or a rule for inferring a fact. For example, the statement:

```
'alice' says 'angry-birds' isGood.
```

Should be read as an assertion by Alice (a principal), that Angry Birds (a constant) is good (a predicate). An example of a rule might be:

```
'alice' says App:A isGood
  if A isFree.
```

This example should be read as an assertion by Alice that any App (a type), referred to as A (a variable) is good if A is also free (a conditional).

As well as assertions about whether variable or constant subjects meet certain predicates, AppPAL can be used to describe delegations, such as:

```
'alice' says 'bob' can-say App:A isGood.
```

[1]Sometimes with minimal edits as some constraints used are not implemented.

Which should be read as an assertion by Alice that if Bob says an App, A, is good then she will also agree that it is good. AppPAL can also describe roles, the assertions:

```
'alice' says 'polygon.com' can-say X isGood.
'alice' says 'justin' can-act-as 'polygon.com'.
```

Are read as first an assertion by Alice that she trusts `polygon.com` (a famous video-game news website) to say whether something, referred to as X, is good; and secondly as an assertion that Alice allows Justin (one of `polygon.com`'s editors) to act as the website.

A full description of AppPAL will be given later in the thesis. AppPAL inherits its evaluation rules and semantics from SecPAL which is described in Section 2.1. AppPAL's syntax, however, differs slightly from SecPAL and includes some conventions for writing policies—these changes are described in Section 3.3.

A key aspect of the mobile ecosystem is *delegation*. The user of a mobile device (typically) first logs on to a Google or Apple account before using the device, which fetches all their data from a server on the internet. Rather than keep account information locally an app may prompt the user to log in, delegating to a third-party (such as Google or Facebook) to manage the account ID. Capturing these trust relationships as policies can help clarify the precise terms for authentication and who each principal trusts to make what decisions. For example, an app might trust Google to manage accounts. Google will only authorise a user if the user has authorised the app, and will only allow the app access to the data the user has explicitly authorised:

```
'app' says 'google' can-say
  'app' canLink(User:U, Account:A).

'google' says App:A canLink(User:U, Account:Acc)
  if U hasAuthorized(A), U hasAccount(Acc).

'google' says User:U can-say U hasAuthorized(App:A)
  if U isAuthenticatedWith(Token), Token isValid.

'google' says User:U can-say App:A canAccess(Data:D)
  if D isOwnedBy(U).
```

Users may install apps manually themselves, but they might also buy and

download apps from one or more app stores. They trust these app stores to sell them *good and safe* apps, and delegate the checking of them to the store. Whereas, in the earlier days of PCs, a user might once have done the check themselves (or at least delegated to an anti-virus (AV) package on their computer) now the responsibility is with the stores. Now most apps come signed either by the developer who created it (in the case of Google's Play Store), the store that sold it (in the case of Amazon's app store) or both (Apple's App Store). These signatures ensure integrity and some measure of authenticity, standing for an assertion that the signer makes that the app is safe to run. For example we may take Apple's signature as an assertion that Apple's vetting process has approved the app is safe to use by its customers. A store may delegate to a third-party app vetting service to assess what apps are safe (Yandex and Aptoide stores), or use their own in-house teams.

Users sometimes recommend apps to each other. We can capture, for example, that Alice may trust Bob to tell her which apps are good.

```
'alice' says 'bob' can-say App:A isGood.
```

Some may consider what apps they want to use on their phone and come up with informally applied personal policies that describe how they want to use them. They may never write these policies down, but they might take the form of preferences that influence the apps they choose, by capturing these we can start to examine and compare policies as well as potentially enforcing them. Bob might recommend any app by Nintendo:

```
'bob' says App:A isGood
  if A isGame,
     A hasDeveloper('nintendo').
```

Bob might trust reviews and review sites to give him an idea of an app's quality.

```
'bob' says App:A isGood
  if A hasReviewScore(N)
  where N > 60.

'bob' says 'metacritic' can-say
  App:A hasReviewScore(Percent:N).
```

Bob might also recommend an app based on its app store categorisation and its permissions.

```
'bob' says App:A isGood
  if A hasCategory('flashlight'),
     A hasPermissions(P)
  where ! contains(P, 'INTERNET').
```

Users allow their employers to say how they should their devices, who may in turn delegate to IT departments, to write rules, which may delegate back to the users to state what rules they're willing to follow.

A company looking to control their employee's mobile devices at work might write a Bring Your Own Device (BYOD) policy that their employees agree to follow. They might also use MDM software to control some aspects of their devices. The company might write these with varying degrees of formality but often they use natural language. This adds vagueness and can lead to confusion about how the company upholds the policy. By describing the policy in a formal language we can express the policy rigorously. We can start to make comparisons between users, and with rules for checking the policy start to help the user to make decisions more accurately, or measure the extent a user follows their stated policy. Using formal languages we can model the policies precisely, helping clarify their meanings and make precise comparisons between different policies. We could tie the rules in the BYOD policies to the MDM tools used to check them.

In the US all software for use with medical data must conform to a policy called HIPAA, and this includes apps on mobile devices. The HIPAA act covers many rules specific to medical software including requirements for keeping medical records confidential. If a company needs its employees to use such apps they could use static analysis tools to check for some aspects of the HIPAA policy. A company might use Mallodroid [47] to detect when apps send data unencrypted.

```
'company' says Employee:E canUse(App:A)
  if A isHIPAAConformant
  where mallodroidCheckSafe(A) = True.
```

It is important not to confuse the tools and techniques we might use to uphold parts of a policy with the end goal of ensuring compliance. A taint-tracking tool, like TaintDroid [46] (for example), can tell when sensitive data is being leaked

out of a network socket. If the end security goal is to prevent employees leaking data, this only covers part of the policy—what about a malicious employee stealing files by printing them and walking out of the office with them? How do we run the tool and when? Using a tool to check for compliance does not necessarily mean that a policy is fully implemented. A formal language that lets us sever the policy from its implementation can help us understand the policy precisely, and show precisely how we check the policy. It lets us see which tools are checking what rules, and identify gaps where the policy is not being checked sufficiently.

These trust relationships and delegations permeate the entire mobile ecosystem. They represent an important aspect of the ecosystem that a policy language should catch to describe the relationships and policies within it.

In this thesis we will come back to these ideas of user's personal privacy policies and BYOD policies as they show two different aspects of how policies are used within the mobile ecosystem. User's privacy preferences are very informal and may not be something a user would ever consider writing down. Rather, a user's privacy preferences guide which apps they might use. A user might even be willing to use an app that does not match their preferences in some cases, such as using the Facebook app (which can access lots of data on a user's device) whilst being generally unhappy sharing their data with their software. In contrast a BYOD policy can exist as a formal agreement (though often *informally* specified) between the company and its employees. An employee might reasonably expect to face consequences if their employer discovers they broke the BYOD rules.

Superficially these policies also resemble classic discretionary access control (DAC) and mandatory access control (MAC) policies. A company *mandates* the BYOD policy, the user uses their *discretion* when picking the apps they use on their phone. An interesting aspect of the mobile ecosystem is that the policies are more complicated than simple MAC and DAC distinctions and use aspects of both. In Chapter 5 we will look at how companies use employee's discretion to judge if aspects of the company's BYOD agreements (such as ethical policies) have been followed.

In contrast, users use their discretion to follow their privacy preferences when picking and using apps. The user must decide whether an app has the need to access certain device functionality or be installed on their phone at all. A

user using a *fine-grained permissions system* (such as one described in Section 2.3) or creating a curated app store (we give a method to do so using a policy in Section 4.3). Now the user will end up writing a MAC-style policy describing how their phone should behave and what apps the store should sell.

BYOD policies and privacy preferences are only a subset of the policies we might see in the mobile ecosystem. We will survey some other policies in the course of the thesis including app store terms and conditions, and the app signing models built into the mobile OSs, but we chose to focus on BYOD and privacy preferences as they covered a range of policy styles, high and low-level policy topics and could showcase some of what makes the mobile ecosystem fascinating.

1.2 Capturing Mobile Security Policies Precisely

The topic of this thesis is how can we capture the informal policies and trust relationships surrounding the mobile ecosystem and use formal languages to model and examine them? Existing research on mobile security policies has focused on enforcing app permissions policies—the *fine-grained* permissions systems. These permission systems allow for new and powerful ways of expressing how users want their devices to behave, but they do not deal with the fact that users say not typically express their policies in terms of permission sets or formal policies but instead might use natural language.

Existing work has given us the mechanisms for enforcing arbitrary mobile security policies, but not the means to link them back to the human natural language informal policies people use in practice. What existing research lacks is the mechanisms to capture and reason about the informal policies precisely, and then link them to the tools and mechanisms that can enforce them.

With the goal of showing how to capture the informal mobile security policies precisely in mind, this thesis attempts to answer the following research questions:

- *How can we capture precisely an informally specified mobile security policy using a formal language?*

  This is the central question of the thesis. We propose using an authorisation logic as a formalisation to capture mobile device policies as these logics

have been previously used to capture policies. Based our requirements for the language and a survey of existing policy languages we suggest using a SecPAL-based language (AppPAL). The rest of the thesis contains various case-studies where we show that our proposed language can capture the policies and give us greater insight into their rules and provide potential enforcement mechanisms.

- *Do we see personal app privacy preferences reflected in users' choice of apps?*

  To answer this question we have access to data about users policies and app installations—therefore we use quantitative research methods as AppPAL gives us a mechanism to query policies against data.

  Existing work has identified 4 generalised app privacy preferences people state they follow [82]. Existing work has also created a data-set of what apps users installed [90]. To answer this question we encode the stated privacy preferences as AppPAL policies, then use AppPAL to find the apps that would be accepted by the policy. The extent a user is following the policy is found by measuring quantitatively the percentage of apps that the user has installed that met the policy as a percentage of the apps they installed overall.

- *What are the common decisions in BYOD policies, and are these the decisions that MDM tools help to enforce?*

  This question lets us use AppPAL for a case-study into BYOD policies. We do not have access to any data about BYOD usage, but we can examine the policies. The research methods used to answer the question are encoding into AppPAL (for the policies) and survey for the functionality of the MDM tools. This allows us to further explore the policies on the basis of our encoding, without having to deal with the ambiguity of natural language policies.

  We take various BYOD policies and express them in AppPAL. This can be somewhat subjective, so care must be taken to capture the style and intent of the original policy and to use a consistent set of predicates between policies. With the policies encoded in AppPAL we can look for common decisions, idioms and trust structures and make comparisons precisely on the basis of the formal version of the policy. We survey the features

of various MDM tools by examining their websites and documentation. Finally we contrast what MDM tools can do with the decisions that BYOD policies want to make.

1.2.1 Thesis Outline and Publications

The rest of this thesis is organised into the following chapters. Some work described has been presented at various conferences, workshops and PhD symposiums through the course of the PhD. We describe the publications, and show where they fit into the various chapters.

For each of our publications the work described was done by the first author: Joseph Hallett (*me*, the thesis author). The second author, Professor Aspinall (my PhD supervisor), provided invaluable suggestions and advice as to where to go with the research as well as extensive editing.

- **Chapter 2: Background.** This chapter describes Becker et al.'s work on SecPAL, and gives an overview of work on other policy languages including XACML and DKAL as well as the fine-grained permission systems used to enforce policies on Android devices.

- **Chapter 3: Instantiating and Evaluating SecPAL.** The next chapter introduces AppPAL as a language instantiating SecPAL to describe the policies of the mobile ecosystem. We introduce the language through examples before showing how we implemented it. We also describe some modifications to the language from SecPAL to make writing policies easier. We conclude by describing our tools for analysing AppPAL policies for satisfiability and redundancy errors.

  Some early examples were taken from our paper:

  – *Towards an authorisation framework for app security checking [55].* This is a PhD symposium paper that describes how we might use SecPAL to model policies in the mobile ecosystem.

- **Chapter 4: App Stores and App Preferences.** Having described AppPAL, this chapter starts to describe the differences between different app stores and survey their different terms and conditions. We use AppPAL to capture descriptions of user's app privacy preferences; and measure the

extent users follow these preferences when selecting apps by comparing with records of user's app installation history. Finally, we describe a tool for generating *curated* app stores on the basis of a policy.

The implementation described, and work on capturing user's privacy preferences is included in our papers:

- *AppPAL for Android [57].* This conference paper describes AppPAL as an instantiation of SecPAL. We present AppPAL evaluation algorithm, and show how to capture user privacy preferences as AppPAL policies. We use the AppPAL versions of the privacy policies to find examples of users following the policies in a user app-installation data set.
- *Poster: Using Authorisation Logic to Capture User Policies in Mobile Ecosystems [56].* This poster presents early work measuring the extent users seem to follow an AppPAL translation of user privacy preferences.

- **Chapter 5: Applying AppPAL to BYOD Policies.** We move from describing *user-centric* policies, to ones companies might want to enforce. We look at how we can capture BYOD policies using AppPAL by looking at 5 BYOD policies (which are included in Appendix A). In capturing the policies, we identify two idioms that existing MDM tooling does not capture. We also describe how AppPAL could be used to enforce a BYOD policy by integrating with existing tooling.

  This chapter encompasses and extends our work presented in:

  - *Capturing Policies for BYOD [59]* This conference paper shows how AppPAL can be used to capture the rules and trust relationships in BYOD policies.
  - *Common Concerns in BYOD Policies [60].* This conference paper looks at BYOD policies and our efforts to find common areas of concern within them.
  - *Specifying BYOD Policies with Authorisation Logic [58].* This PhD symposium paper describes early work capturing BYOD policies with AppPAL. The paper describes how we could use AppPAL to look for common problems, such as completeness.

- **Chapter 6: Future Work.** This chapter describes possible future work, including a probabilistic variant of AppPAL.

1.2.2 Research Contributions

This thesis makes the following contributions:

- I show how SecPAL can be instantiated with predicates and conventions to create a language called AppPAL to describe the policies of the mobile ecosystem.
- I provide an open source implementation of SecPAL and AppPAL. This also contains tooling for examining policies for common problems such as consistency and redundancy. The previous implementation (by the authors of the SecPAL paper [23]) is closed source, and runs only on Windows. Our implementation is written in Java and will run in the JVM and on Android devices.
- I describe a framework for measuring the extent users follow policies with respect to their app choices. Using this framework and a collection of app privacy policies found by surveying users [82], I show that users do not seem to follow them in practice.
- I provide a table summarising the differences in terms and conditions between 5 different app stores.
- I provide a formal version of 5 different BYOD policies, written in AppPAL. Using the formal versions of the policies to compare the contents of the policies I identify two idiomatic forms of BYOD policy rules that have not been examined before or supported by existing tooling.

It is difficult to judge the extent the investigations in this thesis have been a success. We do not, for the most part, provide any proofs that could be checked to show our AppPAL policies are correct, more precise and more powerful than the natural language ones we claim to capture. In fact, the act of writing down the policies in AppPAL requires a degree of subjectivity to capture the style and intent of the original policy. It would be reasonable to look at some of our policies and say that a different version might be more accurate or more descriptive.

As well as describing the policies, this thesis also describes how we can compare and analyse the policies. The AppPAL language give us rules and a grammar for capturing the policies, decisions and trust relationships of the mobile ecosystem. It is, of course, possible to disagree with the way we have captured and presented the policies—but our versions of the policies, and approaches we have taken, can provide a baseline to compare future work against. If future authors can find new policies (hopefully written in AppPAL) that capture the trust relationships differently, in more detail or more precisely, then by comparing our two policies we could see precisely how they differ and gain a greater understanding of the mobile ecosystem.

One approach to judging this work is to consider what the act of capturing the policies in a formal language lets us do. We believe that expressing mobile security policies precisely lets us see the trust relationships more clearly. It allows us to make comparisons on the basis of a formalisation. It helps us understand and argue about the policies. If by capturing the mobile security policies, what we learn is interesting and leads to worthwhile discussion, then our investigations are a success. Alternately if what we have discovered is trivial, wrong, and our language fails to capture the policies in any way, then this work must be judged a failure.

I believe the former is the case. This work is interesting. That my work has lead to publications and informal discussions suggests it is interesting to others too. There is an element of subjectivity in the way we have captured the policies, but I believe the work presented in the rest of this thesis shows that *we do* manage to capture the trust relationships and policies of the mobile ecosystem successfully.

Chapter 2

Background and Related Work

How do computers make policy decisions and how can we capture the computer's decision making process. Suppose Alice, a researcher, wants to run a program to analyse some data—how does the computer decide what she can run and which data she can access? In this chapter we introduce formal policy languages as a mechanism for capturing these decisions, and showing how the decisions can be made. We given an initial example of a researcher attempting to run a program on a computer and illustrate it with SecPAL—a policy language for capturing access control decisions that forms the basis for much of the work in the rest of the thesis. Policy languages (also called *logics of authorisation*) describe rules for when to allow certain actions. We go on to describe past and current work on policy and access-control languages, as well as work on fine-grained permission systems that can be used to implement policies on mobile devices.

2.1 SecPAL

The scenario in this section are based on an example given by Becker in the original SecPAL paper [23], but extended and illustrated by us.

Suppose Alice, who works as a researcher, wants to run a program on her computer to analyse some data. How does her computer decide whether she is allowed to do this? The answer is that her computer has a *policy*—a set of precise rules and a decision procedures that describe *who* is allowed to do *what*. When Alice makes the request to run the program the computer checks whether its policy permits Alice to run things and if so, runs the program as

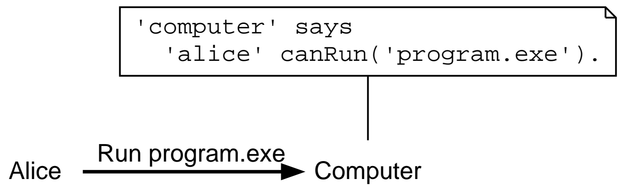


Figure 2.1: Simple example of a policy scenario.

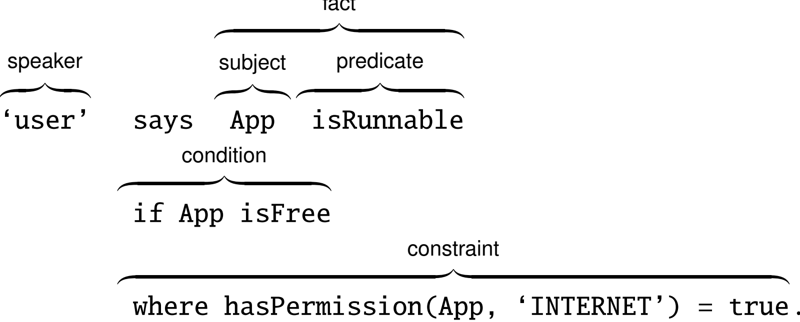


Figure 2.2: Structure of a SecPAL assertion.

Alice requested.

A simple example of this might be Alice running programs on her own personal computer. Her computer has the simple policy that Alice can run her specific program. This is illustrated in Figure 2.1. Alice send a request to the computer to run her program (the bold arrow) and the computer has a policy that allows it to make decisions (the note).

```
'computer' says 'alice' canRun('program.exe').
```

The policy is written in SecPAL (the *Sec*urity *P*olicy *A*uthorization *L*anguage): an authorisation language developed by Becker, Fournet and Gordon to describe policies and delegation chains for distributed systems [23]. SecPAL will be explained by example through out the chapter, but in summary SecPAL is a high-level human-readable language that separates policy specification and maintenance from the implementation mechanisms. A summary of the different parts of an assertion (a single SecPAL rule or statement) is shown in Figure 2.2, and a BNF-description of SecPAL's grammar is shown in Figure 2.7.

When Alice sends her request what does the computer do? It takes Alice's request and constructs a query from it, and then checks the query against the policy. If the query is valid with respect to the policy then the computer allows Alice's request. Written in SecPAL Alice's query is:

```
AC,∅ ⊢ 'computer' says 'alice' canRun('program.exe').
```

Assertions in SecPAL are made by an explicit *entity* called the *speaker* or *principal* (the `computer` in this case) who says a fact about another entity called the *subject* (`'alice'`). The subject of an assertion may also be a speaker, or may be a voiceless entity the speaker says something about (the program, for example). In SecPAL the speaker of an assertion represents the original entity making the decision—assertions can be shared between principals (which we will come to in Section 2.1.1) but in this case since it is the computer's rule about what programs Alice can run, it is the computer who makes the assertion.

The computer collects all of it's policy rules into a structure called an *assertion context* (AC), and attempts to decide whether the AC can be used to construct a proof that the query q is true (under a possible variable substitution θ, which in this case is the empty set as there are no variables).

$$\overbrace{AC}^{[\texttt{'computer' says 'alice' canRun('program.exe').}]}, \theta \vdash \underbrace{q}_{\texttt{'computer' says 'alice' canRun('program.exe').}}$$

SecPAL's proof theoretic semantics are given in Figure 2.9, and the derivation rules it uses to decide is a particular assertion holds or not are given in Figure 2.8. SecPAL has two main forms of decision procedure: The $\vdash$ form is used to evaluate a query with respect to an assertion context, and the $\models$ form is used to decide whether, given an assertion context and a flag indicating whether delegation is allowed, SecPAL's derivation rules can be used to derive the assertion from the assertion context.

```
'computer' says
  User canRun(Program)
  if User isLoggedIn,
     Program isExecutable.

'computer' says 'alice' isLoggedIn.

'computer' says 'program.exe' isExecutable.
```

Alice —Run program.exe→ Computer

Figure 2.3: A policy scenario involving variables.

In this case, the proof Alice's computer constructs is a simple application of the *cond*-rule (Figure 2.4). The *cond*-rule states that a queried-assertion is valid with respect to an assertion context if:

1. the assertion context contains an assertion that has the query as its head (possibly under a variable renaming θ),
2. all the conditionals in the body of the assertion are also valid with respect to the assertion context,
3. any constraints are satisfied,
4. the fact in the head of assertion contains no variables (after the renaming θ is applied).

The computer's query satisfies the policy, so it allows Alice to run her program. This example is trivial: the policy contains one hard-coded rule and it states explicitly that Alice can run the program (and there are no variables or constraints). We can (and will) show much more complex policies that capture more subtle behaviours and decisions, but this demonstrates the basic problem policy languages aim to solve. Given a set of rules, when a system is asked to make a decision what should it do and how can it show that the rules were followed? The proof shows how the policy was evaluated, and acts as evidence that the policy was followed.

Of course, SecPAL can express more complex policies than a simple hard-coded rule. A more realistic example might be that the computer allows anyone who is logged-in to run programs that are marked as executable (Figure 2.3). The policy now becomes:

```
'computer' says User canRun(Program)
  if User isLoggedIn,
    Program isExecutable.

'computer' says 'alice' isLoggedIn.
'computer' says 'program.exe' isExecutable.
```

This introduces *variables* that allow entities to be generalised. Instead of the rule specifying that Alice can run a specific program, Alice is replaced by a variable *user* (who must be logged in) and the program is also replaced by a variable with the condition that the program must be executable. Two additional assertions are added to the policy: the first stating that Alice is logged in, and the second stating that her program is executable. The proof (Figure 2.5) becomes more complex, but it still gives us a structured way of showing *how* a decision was made.

While assertions in an assertion context are allowed to contain variable (subject to rules), SecPAL queries are not allowed to contain free variables. Becker described an *IN/OUT* query safety condition (Figure 2.11) that ensured that for any assertion context the set of possible answer substitutions would be finite. A later extension to SecPAL added guarded universal quantification of variables, however [22]. The extension allowed queries with a ∀ in them, but also required that the query include a *guard*—a statement to restrict the scope of the free variable. For example, a query to search for all apps (known by Alice) that that were recommended by Bob could be written:

$$\forall \texttt{X} \left(\overbrace{\texttt{Alice says X isApp.}}^{\text{Guard}} \Rightarrow \underbrace{\texttt{Bob says X isRecommended.}}_{\text{Guarded Query}} \right)$$

In this query the `Alice says X isApp.` is the guard, restricting the scope of `X` to the values that Alice says are apps, and the `Bob says X isRecommended.` is the guarded query that provides the result.

In the last example the knowledge of who was logged in, and what was executable was written in the computer's policy. Not all information can be

> AC, inf $\models$ 'computer' says 'alice' canRun('program.exe').
>
> ---
>
> ('computer' says 'alice' canRun('program.exe').) $\in$ AC
>
> $\models \top$ (no constraint)
>
> vars('alice' canRun('program.exe')) $= \emptyset$

Figure 2.4: A Simple SecPAL proof, presented in the Fitch style.

> AC, inf $\models$ 'computer' says 'alice' canRun('program.exe').
>
> ---
>
> ('computer' says User canRun(Program) if $\cdots$.) $\in$ AC
>
>> 'computer' says User isLoggedIn.(User $\rightarrow$ 'alice')
>>
>> ---
>>
>> 'computer' says 'alice' isLoggedIn. $\in$ AC
>>
>> $\models \top$
>>
>> vars('alice' isLoggedIn.) $= \emptyset$
>
>> 'computer' says Program isExecutable.(Program $\rightarrow$ 'program.exe')
>>
>> ---
>>
>> 'computer' says 'program.exe' isExecutable. $\in$ AC
>>
>> $\models \top$
>>
>> vars('program.exe' isExecutable) $= \emptyset$
>
> $\models \top$
>
> vars('alice' canRun('program.exe')) $= \emptyset$

Figure 2.5: A more complex SecPAL proof involving variables.

```
'server' says
  User canRun(Program)
  if User isUser,
     Program isExecutable
  where timeSpent(User) < timePaidFor(User).

'server' says 'alice' isUser.

 'server' says 'program.exe' isExecutable.
```

Alice —Run program.exe→ Server

Figure 2.6: A policy scenario involving constraints.

captured by adding statements to a policy. Dynamic information, like time, can be hard to encode for example. Suppose Alice isn't using her personal computer but a cloud server run by a third party. Alice has purchased some compute time on the server, and once her time is up she can't run any more programs. The scenario is illustrated in Figure 2.6.

SecPAL captures policies that rely on dynamic, external information by using *constraints*. A constraint is an arbitrary function that can return some value (such as an entity, boolean, number or string) that is separate from the decision procedures in SecPAL. In Figure 2.6, the `timeSpent` and `timePaidFor` constraints return numbers. So long as Alice has spent less time than she's paid for she is allowed to run her program. How the constraints determine these times this is opaque to the policy: they may go off and query a payment or usage tracking system, but from the point of view of the policy this is a black-box. SecPAL has other mechanisms for handling delegated decisions that allow for greater control over how delegation takes place within SecPAL, but in this case we do not know how the functions work. The only limitation SecPAL places on constraints is that if they contain variables, those variables must have been removed (through renaming in the *cond*-rule) by the time the constraint is evaluated.

AC	::=	$\text{assertion}_1 \ldots \text{assertion}_n$	*(assertion context)*
assertion	::=	e `says` claim.	
e	::=	`X`	*(variables)*
	|	`'A'`	*(constants)*
predicate	::=	`has` | `can` | ...	*(predicates)*
D	::=	`0`	*(no further delegation)*
	|	`inf`	*(unbounded delegation)*
vp	::=	predicate(`(`e_1`,` ... e_n`)`)?	*(verb phrase)*
	|	`can-say` D? f	*(delegation of fact)*
	|	`can-act-as` e	*(role assignment)*
f	::=	e vp	*(fact)*
claim	::=	f (`if` $\text{f}_1, \ldots, \text{f}_n$)? (`where` c)?	
c	|	e'_1 `=` e'_2	*(constraints)*
	|	e'_1 `<` e'_2	*(constraints)*
	|	...	
e′	::=	e | function($\text{e}_1, \ldots \text{e}_n$)	
	|	`true` | `false`	*(booleans)*
	|	integer	*(numbers)*

Figure 2.7: BNF description of SecPAL as used in this thesis. As originally described by Becker, SecPAL is hard to type as it uses subscripts and infinity symbols. We replace these with ASCII equivalents, and allow a missing D in the `can-say` statement (equivalent to `can-say 0`). Terminals are shown in red.

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

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

$$\frac{AC, D \models a \;\texttt{says}\; B \;\texttt{can-act-as}\; c \quad AC, D \models A \;\texttt{says}\; C \;\text{verb-phrase}}{AC, D \models A \;\texttt{says}\; B \;\text{verb-phrase}}\;\text{can-act-as}$$

Figure 2.8: The derivation rules used to evaluate SecPAL. All SecPAL rules are evaluated in the context of a set of other assertions AC as well as an allowed level of delegation D which may be `0` or `inf`. The $\models$ symbol is used by Becker [23] to describe the derivation rules instead of the more usual $\vdash$. This is in order to distinguish SecPAL's derivation rules, from SecPAL's query rules (given in Figure 2.9).

Concepts	$AC, \theta \vdash q$	Defining relation. A query assertion q is valid given the assertions contained in the assertion context AC and a variable substitution θ.
	ϵ	The empty substitution.
Definitions	1. $AC, \theta \vdash \underbrace{e \text{ says } fact}_{q}.$	if $AC, \texttt{inf} \models e\theta \texttt{ says } fact\theta.$ and $dom(\theta) \subseteq vars(e \text{ says } fact)$
	2. $AC, \theta_1\theta_2 \vdash q_1, q_2$	if $AC, \theta_1 \vdash q_1$ and $AC, \theta_2 \vdash q_2\theta_1$
	3. $AC, \theta \vdash q_1$ or q_2	if $AC, \theta \vdash q_1$ or $AC, \theta \vdash q_2$
	4. $AC, \epsilon \vdash$ not(q)	if $AC, \epsilon \nvdash q$ and $vars(q) = \emptyset$
	5. $AC, \epsilon \vdash c$	if $\models c$

Figure 2.9: SecPAL's proof-theoretic semantics as described by Becker [23].

Let $\texttt{'a'}$ $\textit{says fact if fact}_1 \cdots \textit{fact}_n$ $\textit{where c.}$ be an assertion.
A **variable** $\texttt{X} \in vars\,(\texttt{fact})$ is *safe* if:

$$\texttt{X} \in vars\,(\texttt{fact}_1) \cup \cdots \cup vars\,(\texttt{fact}_n)$$

The **assertion** $\texttt{'a'}$ $\textit{says fact if fact}_1 \cdots \textit{fact}_n$ $\textit{where c.}$ is safe if:

1. (a) If $\texttt{fact}$ is flat (it is not a can-say fact), all variables in $vars\,(\texttt{fact})$ are safe.
   (b) If $\texttt{fact}$ is not flat (it is of the form $\texttt{E}$ $\textit{can-say fact}'$), $\texttt{E}$ is a constant or safe variable.
2. $vars\,(c) \subseteq vars\,(fact) \cup vars\,(\texttt{fact}_1) \cup \cdots \cup vars\,(\texttt{fact}_n)$
3. $\texttt{fact}_1 \cdots \texttt{fact}_n$ are all flat.

Figure 2.10: SecPAL's assertion safety conditions.

$$\frac{\exists O \text{ such that } \emptyset \Vdash q : O}{q \text{ is safe}}$$

$$I \Vdash q : O \begin{cases} q \text{ is the query.} \\ I \text{ is the set of ground variables before evaluating } q. \\ O \text{ is the set of newly ground variables after evaluating } q. \end{cases}$$

$$\frac{fact \text{ is flat}}{I \Vdash e \text{ says } fact : vars\,(e \text{ says } fact) - I} \qquad \frac{vars(c) \subseteq I}{I \Vdash c : \emptyset} \qquad \frac{I \Vdash q_1 : O_1 \quad I \Vdash q_2 : O_2}{I \Vdash q_1 \text{ or } q_2 : O_1 \cap O_2}$$

$$\frac{I \Vdash q : O \quad vars\,(q) \subseteq I}{I \Vdash \text{not}(q) : \emptyset} \qquad \frac{I \Vdash q_1 : O_1 \quad I \cup O_1 \Vdash q_2 : O_2}{I \Vdash q_1, q_2 : O_1 \cup O_2}$$

Figure 2.11: SecPAL's IN/OUT query safety condition.

2.1.1 Delegation in SecPAL

In the prior examples, Alice was trying to run a program on just one computer. There was only one policy, and only two entities: Alice and the computer she was trying to run her program on. What happens when the scenario becomes more complex? What happens if there multiple entities, each responsible for their own decisions? How do we capture delegation and trust using a formal language?

A key feature of SecPAL is the ability to delegate statements. SecPAL was designed to make access control decisions over large networks. Rather than have a centralised decision server make all decisions, SecPAL allows the sharing of information through assertions signed by their speaker. This allows different principals to take responsibility for different decisions and make decisions independently.

This time instead of Alice attempting to run a program on her own computer, she will attempt to run a query on her university's cluster using data from a file-server, and by interacting with an HR department who know whose who in the department (Figure 2.12).

Alice requests the cluster run a search on her data. Her data is on the file-server. The cluster has a policy that only researchers can run the search. It also has a rule that says the human resources department (HR) can say who is a researcher.

```
'cluster' says X canRun('grep')
  if X isResearcher.

'cluster' says 'hr' can-say
  X isResearcher.
```

The cluster queries HR if Alice is a researcher. HR responds by saying she is.

```
'hr' says 'alice' isResearcher.
```

The cluster does not know how HR knows that Alice is a researcher. But it is content *to trust* HR's assertion that she is. HR may have a SecPAL instance and policy of their own to make this decision and send it to the cluster. Alternatively they might be using a conventional database. Provided they give this SecPAL assertion to the cluster, it does not care how they came by it. The one limitation the cluster has is that it *must* be HR telling them. HR cannot delegate the

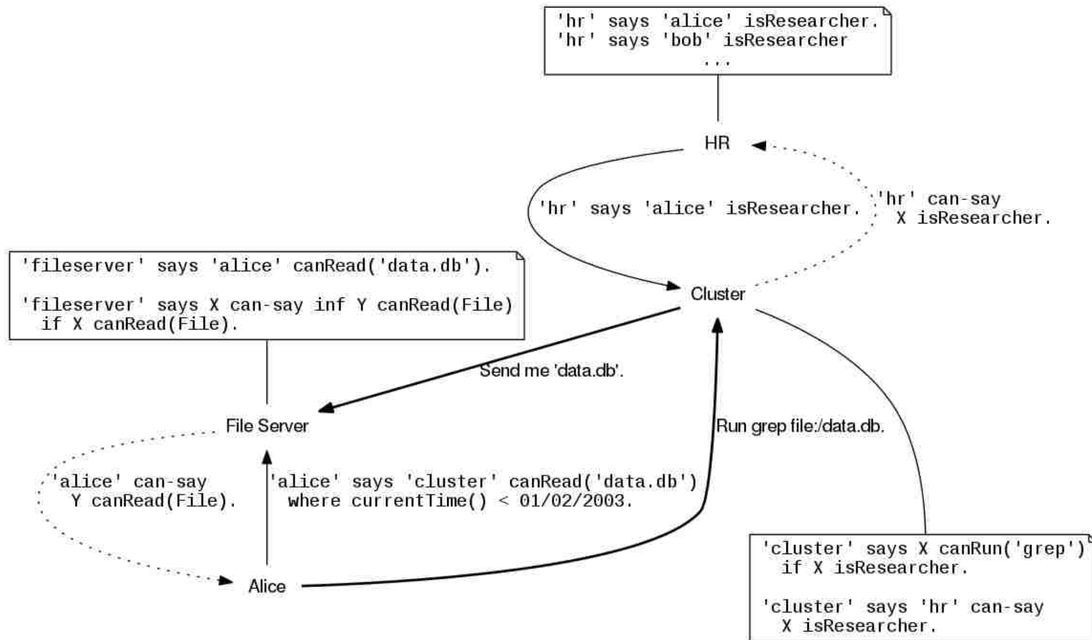


Figure 2.12: Example of delegation when running a command on a cluster. Bold links show requests, plain links show the sending of SecPAL statements, and dotted links indicate delegation relationships. SecPAL assertions at each location are shown in notes.

decision further.

The cluster is now convinced that Alice may run the search. It requests the database from the file server. The file server knows that Alice can read her data and that anyone who can read a file may say who else can read it.

```
'fileserver' says 'alice' canRead('data.db').
'fileserver' says X can-say inf Y canRead(File)
  if X canRead(File).
```

Using SecPAL, the file server determines that Alice can say who reads her data. Alice gives the file server a statement that the cluster can read her file (for a short time period).

```
'alice' says 'cluster' canRead('data.db')
  where currentTime() < 01/02/2003.
```

The file server gives the cluster the data. The cluster runs the search and hands the results back to Alice.

This simple example shows how different principals can make decisions using delegation mechanisms. SecPAL allows for more complicated delegation

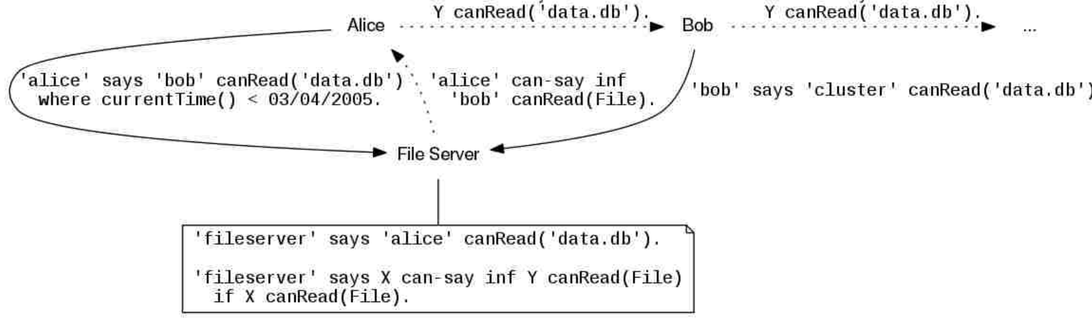


Figure 2.13: Example of unbounded delegation.

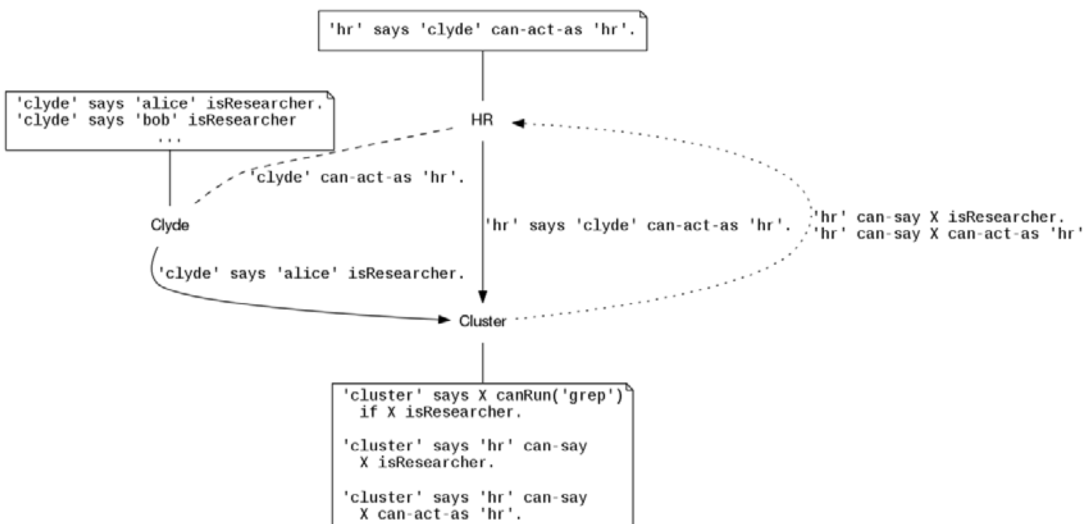


Figure 2.14: Example of delegation with roles. Role relationships are shown with dashed links.

relationships, however. The file server gave Alice the ability to delegate who could read her file by using the `can-say inf` verb. Alice might allow Bob to share her data set with others who might also be allowed to share it for a limited time (Figure 2.13).

An alternative to asking to the HR server directly if Alice is a researcher is to use roles (shown in Figure 2.14). Many people work in HR. The cluster might accept the word of any of the people who work there. To do this the cluster delegates to HR to name anyone who *acts as* HR. Suppose that HR responds that Clyde can act for them.

```
'cluster' says 'hr' can-say
  X can-act-as 'hr'.


'hr' says 'clyde' can-act-as 'hr'.
```

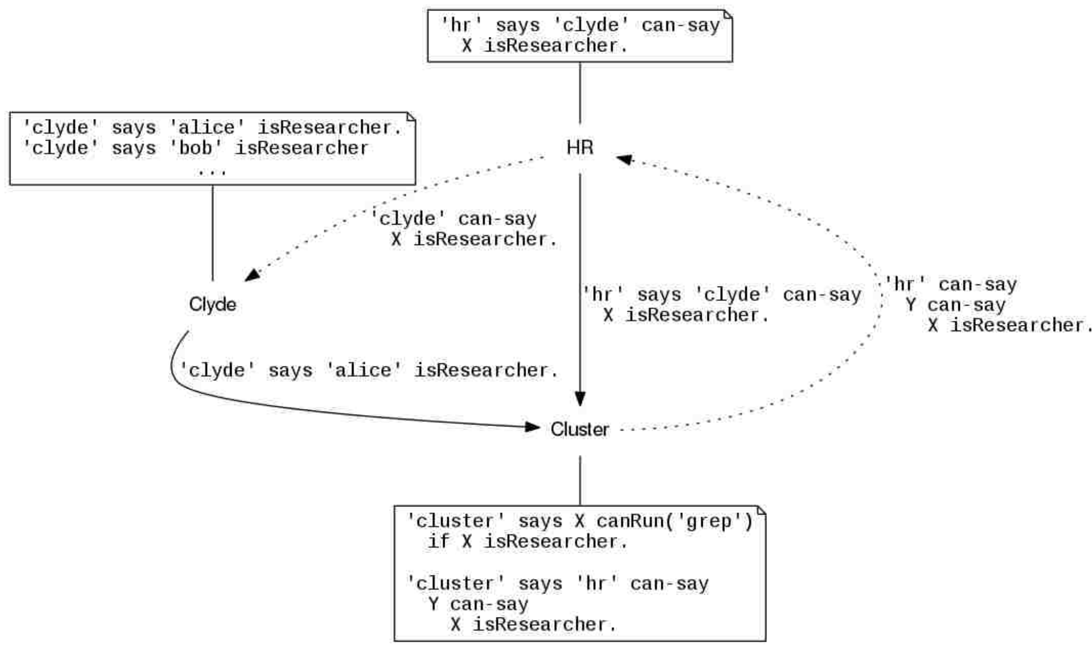


Figure 2.15: Example of depth-bounded delegation.

Now, on the cluster, Clyde's word is as good as HR's. Clyde sends the necessary facts about Alice. The policy check now runs as before. Note that the restrictions on HR also apply to Clyde: he still cannot delegate the decision further. Clyde may have more capabilities than HR if there are other provable assertions about him. The *can-act-as* means that Clydes is at least as powerful as HR: he inherits HR's capabilities.

Depth-bounded delegation with the *can-say* statement is an alternative to roles (Figure 2.15). Instead of letting HR state who is a researcher to HR, the cluster lets HR *decide who decides*. HR delegates to Clyde and the process continues as before. Depth-bounded delegation allows delegation statements to be chained to an arbitrary (but finite) depth, without allowing for unbounded delegation, as unbounded delegation can become problematic if *loops of trust* are introduced. Loops of trust, as shown in Section 2.1.1, when there is a chain of delegation between principals that forms a loop. SecPAL does not prohibit setting up loops like and it is left to the implementation to detect and resolve them, as they can cause the derivation rules to hang.

It is preferable to roles as it allows HR to delegate some but not all decisions to others. If role assignment is used then, on the cluster, anywhere `'hr'` follows the `says` in an assertion, then it can be replaced with `'clyde'`: they are the same.

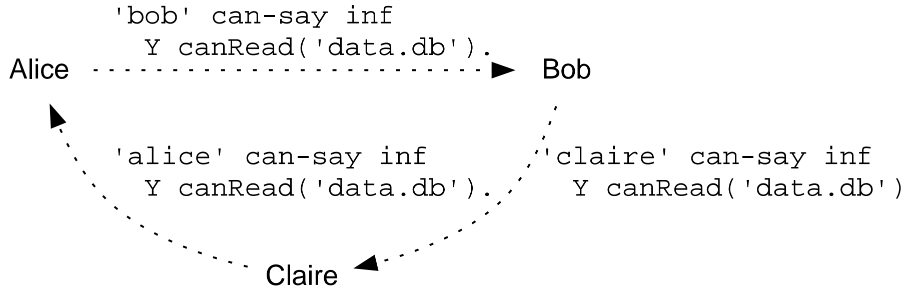


Figure 2.16: Delegation with a loop of trust. Alice trusts Bob who trusts Claire who trusts Alice in turn to make a decision.

SecPAL's delegation mechanisms can describe many trust relationships between entities, as we have described above. Yet it remains conceptually and semantically simple, by using just the `can-say` and `can-act-as` phrases to capture them. This power makes SecPAL attractive for situations where entities are distributed and there is no central decision point, as we can capture the trust relationships between entities and at different places. This makes SecPAL a very appropriate language to model the policies of the mobile ecosystem, as every device, user, company and store has their own policies and makes their own decisions, sometimes by talking to each other. We will discuss our reasons for using SecPAL to capture the policies of the mobile ecosystem further in Chapter 3, but to summarise its ability to delegate and capture policies makes it a good starting point.

2.2 Policy Languages

We introduced SecPAL to describe policies, and will study it further in the rest of the thesis to describe the policies of the mobile ecosystem. Other policy languages exist, however, and in this section we give a survey of some other influential policy languages. A timeline of the development of the languages mentioned here is shown in Figure 2.17.

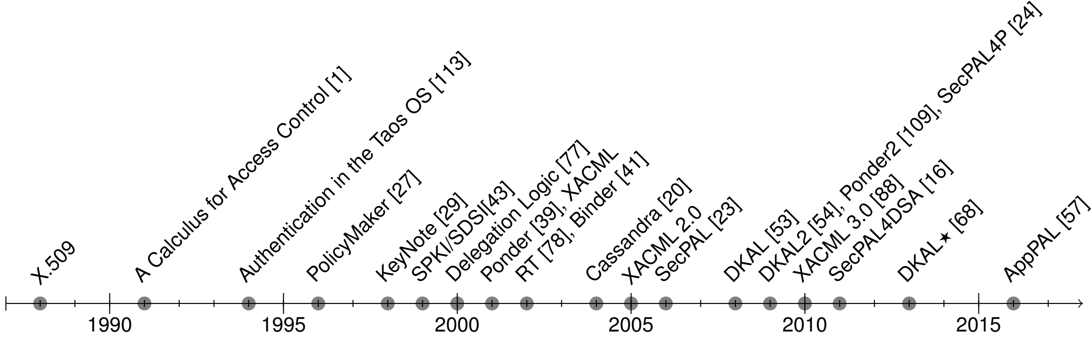


Figure 2.17: Timeline of the development of different policy languages.

PolicyMaker. PolicyMaker [27] is a language for permitted actions. It grew out of the logics of authentication of Wobber, Abadi, Burrows and Lampson [113, 1]; as well as a dissatisfaction with identification mechanisms such as X.509 and PGP certificates. These mechanisms identified users but could not link them to what they could do.

To describe the authorisations, assertions hold trust information. Many assertions form a policy. An assertion has a source (either the local policy document or a public key). The source *asserts* that the holder of a key (or more complex arrangements such as three of four keys) can do any action that matches the *filter*. The filter itself is an arbitrary program that can make a yes/no decision. Blaze et al. give examples using regular expressions and a reduced version of AWK [4]. They note, however, that any programming language could be used. The device policy can be queried by a key holder *requesting* a given action.

To integrate PolicyMaker into a real system, software would have to be modified to automatically send PolicyMaker queries to a central query server in response to events, in order to decide what to do next. Blaze et al. give an example of this scheme being used as part of an email server. A user, Alice identified by key `0x12345678`, can send emails with the `from` header set to Alice and the organisation set to Bob Labs.

PolicyMaker is installed on a server. It receives queries and gives answers. The policy installed on the server in this example would be:

```
policy ASSERTS
  pgp:'0x12345678'
  WHERE PREDICATE=regexp:'(From: Alice) &&
    (Organization: Bob Labs)';
```

When Alice sends an email using a PolicyMaker enhanced SMTP client she signs her message with her key. The mail server checks the signature and queries the policy server with her message:

```
pgp:'0x12345678'
REQUESTS 'From: Alice
        Organization: Bob Labs

        Hello World!';
```

If Alice's message is okay, then the SMTP server will send it. If it does not match the policy, then it will not.

KeyNote. PolicyMaker is the basis for KeyNote [31, 29]. KeyNote simplifies the arbitrary filter languages, offering instead one based on C that always returns a boolean answer. KeyNote allows the policy server to do the signature verification instead of the querying application. It also makes the syntax more readable. KeyNote also trades PolicyMaker's generality for a more realistic scenario using public-key infrastructure. The prior policy for KeyNote could be written:

```
Authorizer: "POLICY"
Licensees: "RSA:abc123"

KeyNote-Version: "2"
Local-Constants: Alice="RSA:12345678"
Authorizer: "RSA:abc123"
Conditions: (app_domain == "RFC822-EMAIL") &&
   (name="Alice") &&
   (organization="Bob␣Labs");
```

As with PolicyMaker, the integration of KeyNote into any application is left to developers. Consequently the automation of any queries (and deciding what to do with whatever results KeyNote may return) is left to the application. Blaze et al. note that [28]:

> "KeyNote does not directly enforce policy it only provides advice to the applications that call it. In other words, KeyNote assumes that the application itself is trusted and that the policy assertions it specifies are correct. Nothing prevents an application from submitting misleading or incorrect assertions to KeyNote or from ignoring KeyNote altogether."

Checking whether a PolicyMaker or KeyNote policy is satisfied is NP-hard [30]. PolicyMaker assertions can involve arbitrary programs written in Turing complete languages. Blaze et al. restrict these programs to those that run in polynomial time for all inputs pertinent to a request.

A weakness of these languages is that they cannot express relationships among users. You cannot have policy where the subject is a set of users. The example policy could not be written as *anyone working in R&D can send email from Bob Labs.*

SPKI/SDSI. Unlike PolicyMaker, SPKI/SDSI [43] was designed to associate keys with roles. A user, Alice with key K_A, can present a certificate that says she **can act as** (a role assignment) a *Bob Labs employee* (authorised by Bob with key K_B) for one year.

(K_A, `BobLabsEmployee`, K_B, 1 year)

Bob can also create authorisation certificates to let his employees to send emails. Optionally they can delegate the decision further.

(K_B, (K_B, `BobLabsEmployee`), $\bot$, `send_email`, 1 year)

PolicyMaker checks whether to allow a specific user's action. SPKI/SDSI associates users with roles and roles with tasks. The SPKI/SDSI version of the email sending policy (shown above) does not specify that all emails sent by Employees must have a field listing the lab as the sending organisation. That part of the policy must be added by whatever implements the `send_email` functionality. One of the advantages of SPKI/SDSI is that it allows a higher-level view of the policy by associating groups of users with a role, and roles with allowed actions, rather than specifying the precise mechanism of the checks.

SPKI/SDSI does not have a formal semantics. The language's precise meaning must be inferred from RFC 2693, which loosely defines it in English [43]. There are several later attempts at fitting semantics to the language [71, 2, 66, 34]. Not all covered every aspect of SPKI/SDSI's features, however. No definitive standard has appeared.

RT. The focus on roles led to the RT family of languages [79]. RT associates policy decisions with roles. This is similar to how Role Based Access Control (RBAC) systems associate access decisions to the roles a user holds. Policies are expressed as Horn clauses. A rule such as:

```
Bob.employee :- alice.
Bob.employee.sendEmail :- Bob.employee.
```

is read as *Bob says Alice is an employee, and Bob says an employee can send emails if they are an employee.* Li et al. describe many variants of RT with various features. The most basic variant is RT_0 [81]. RT_1 adds support for parameterised roles. RT_2 adds logical objects on top of the roles. As well as these variants, the RT family of languages supports optional feature sets: RT^T allows for policies with thresholds (i.e. Alice can email someone if two out of three of the board members of her company approve it). RT^C adds constraints. Finally, RT^D adds delegation [79].

Unlike PolicyMaker, the RT family of languages is tractable. Li et al. prove a guarantee that a query will be answered soundly in polynomial time in the size of the policy. To give this guarantee, the researchers showed the languages could be reduced to Datalog. Datalog is a database language with known complexity guarantees and fast evaluation. Datalog is similar Prolog, but without negation, complex arguments, and the `is` statement. They also showed similar languages, like Binder [41] and Delegation Logic [80, 77], could be described using Datalog as well.

Datalog is limited in that it cannot describe structured data. Consider a rule that lets Alice send email between 9am and 5pm. We might want some function to compare whether a time is within a range. In Datalog we cannot trivially write this function. We would have to say for each possible time if it is in that period. Generally, when there is data that has structure such as file paths, times or numeric intervals: Datalog databases can become overly large.

To solve this Li et al. modified Datalog to create Datalog^C [76]. Datalog^C is based on Constraint Datalog [98, 99] and supports constraints whilst keeping Datalog's tractability. It focuses on the constraints typical to a policy languages (such as constraints for handling files and directories) instead of those for database programming.

Showing a policy language is translatable to Datalog^C allows the language author to prove that evaluating policies in their language can be done with the same time and space complexities as Datalog^C. Datalog^C is used as a foundation for many other policy languages including SecPAL as well as RT.

Ponder. Ponder, like SecPAL, is a language for specifying policies for distributed systems [39]. Ponder supports positive and negative authorisation, delegation, obligation, roles and constraints. It uses constraints to extract the attributes of principals, read state, and deal with time. A policy that trainee engineers are not allowed to send emails could be written:

```
inst auth- engineersCanEmail {
  subject e =/Engineer;
  target /mail_server;
  action send_email();
  when e.status() == ‘‘trainee’’;
}
```

Since Ponder allows positive and negative policies, conflicts can occur. Ponder does not resolve the conflicts itself. Instead, it detects them using static analysis and reports them to the policy author as bugs in the policy specification. Ponder2 [109] simplified Ponder and added policies for decentralised environments.

Ponder2 also provides a control language called *PonderTalk*, which lets the decision engine send messages to the objects it manages, as well as the objects send messages between themselves. This allows for the automation of some queries in response to PonderTalk message being sent, as well as the management of obligations by allowing different objects to notify others of what they *must* do.

Cassandra. Cassandra is a trust management language to model large systems. It was demonstrated on the NHS Spine: a system to let healthcare workers share patient data [20, 21]. The language is similar to RT, using both a Datalog^{C} and some of its syntax. Cassandra, however, focuses on roles instead of general access control mechanisms. Principals *activate* roles when they need to act in its capacity. This lets one principal hold different roles, with different capabilities, at different times and at different locations. A *hospital admin* might allow a *clinician* access to a *patient*'s records if they have the role of *Clinician* at the hospital, and can *activate* the role of *Treating Clinician* for that patient at this *hospital*:

```
hospital@admin.permits(clinician, Read-Records(patient)) <-
  hospital@hasActivated(clinician, Clinician(hospital)),
  hospital@canActivate(clinician, TreatingClinician(hospital, patient))
```

A prototype of Cassandra was implemented allowing it to answer some of the queries associated with requesting electronic health records, but this wasn't further developed (that we know of) into an automated system for supplying health records.

Cassandra allows for powerful control of different roles, and enables delegation by allowing third-parties to activate and remove roles on others. Becker worked on Cassandra for his doctorate. He went on to work on SecPAL.

DKAL The DKAL family of languages [68, 53, 54] grew from SecPAL as a policy language for a distributed system's interacting agents [26]. Assertions are communicated as signed statements called *infons* between principals. For example Alice might wish to recommend Bob an app. Alice, therefore, sends to Bob the infon:

```
alice said com.rovio.angrybirds is a good app.
```

Bob is free to do with this information as he wishes. He might accept app recommendations from anyone and may have a rule to learn facts:

```
with M: String, P: Principal
  upon
    P said M is a good app
  do
    learn P said M is a good app
```

DKAL2 [54] simplifies DKAL language removing SecPAL's constructs for roles (the *can-act-as* statement) as the authors of DKAL and DKAL2 could not find a use for them in practice. It also adds support for sending infons if the recipient has completed an action: for example only sending an app recommendation if the recipient has asked for it. DKAL* [68] took the ideas of information sharing further and showed how to create executable protocols from DKAL programs.

DKAL improves over SecPAL by giving a means for transferring and reacting to information. These might be helpful to describe how protocols worked within the mobile ecosystem and for expanding how obligation policies would be triggered. The authors of DKAL showed that all SecPAL policies could be

expressed in DKAL [53].

The work on AppPAL in this thesis is based on SecPAL rather than a DKAL-based language. As DKAL extends SecPAL, and is backwards compatible with SecPAL, AppPAL could gain, if needed, increased expressiveness and the ability to describe protocols just by switching interpreters. For the work described in this thesis however, we did not require the additional expensiveness.

XACML XACML is a policy language with an industrial standard [88] that can be extended to fit many scenarios. It is an attribute based policy language, but can also describe role-based policies. It has tooling and enforcement infrastructure available from Oracle. The third version of the language (which supports delegation) was released just after we started our own work on AppPAL. As a standard, it is important to summarise and briefly describe why we chose to base our own work on SecPAL, a comparatively obscure policy language, instead of the more well-known XACML.

XACML policies are expressed as sets of *rules* that describe whether a specific action should be allowed or denied. When making a *request* if a rule's target matches it then the appropriate action should be taken. Rules are combined into policies containing many rules which can contradict each other if multiple ones match a given request. To handle contradictions a *combining algorithm* is used to decide how to proceed. Typical algorithms include:

Permit-overrides. If a rule gives a *permit* result, then the request is permitted.

Deny-overrides. If a rule gives a *deny* result, then the request is denied.

First-applicable. The rules have an order of precedence. The first rule that gives a result decides the outcome.

Only-one-applicable. Only one rule may match. If there are multiple matches, then an error is returned.

XACML's designers used natural language to describe the semantics of XACML. This makes the semantics notoriously difficult to interpret [94]. There have been several attempts to describe XACML's semantics formally [95, 96, 33]. Some of these translations only describe a subset of XACML's syntax [62]. Others describe older versions of XACML which are not compatible with the

latest language versions [3], Others have been found to contain mistakes caused by ambiguities in the XACML specification [32, 62]. All these attempts help us understand XACML, but the lack of a single standard semantics make it less attractive to extend to a new domain. As the language grows and changes there are no present guarantees that any of the formal semantics will remain correct and applicable to the current version of XACML. In contrast, SecPAL's semantics are given precisely by Becker [23].

Whilst SecPAL was designed for readability, XACML policies are verbose and difficult to read. For example, to describe a simple policy that grants a principal, *Alice*, permission to do whatever she wants we might write the policy shown in Figure 2.18.

To help developers write policies, alternative notations are available that compile into XACML's XML notation. ALFA is an alternate notation for XACML [89] maintained by the XACML developers. The equivalent ALFA policy of the above *allow Alice all* policy would be:

```
namespace alice {
  policy policy {
    target clause Attributes.subjectId == ”alice”
    apply permitOverrides
    rule {
      permit
    }
  }
}
```

The ALFA policy is more concise and shows the policy's meaning more clearly than the XACML version. There is an (Eclipse based) compiler for ALFA policies into XACML [14], but there are not tools for translating XACML to ALFA. This means that any XACML policies would need to be rewritten to take advantage of ALFA's syntax, and that any tweaks made to a XACML policy cannot be trivially reintegrated into the ALFA version. Furthermore, any analysis of XACML's semantics cannot be trivially used to understand the semantics of ALFA.

Other notations exist including graphical languages [86], languages based on propositional logic [116] and answer set programming [95].

Whilst XACML 3.0 does support delegation [87], earlier versions do not. The XACML 3.0 standard was published in 2013, after deciding to start work with

```
<xacml3:Policy xmlns:xacml3="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
  PolicyId="http://axiomatics.com/alfa/ identifier / alice . policy "
  RuleCombiningAlgId="urn:oasis:names:tc:xacml:3.0:rule-combining-algorithm:permit-
      overrides"
  Version="1.0">
  <xacml3:Description />
  <xacml3:PolicyDefaults>
    <xacml3:XPathVersion>http://www.w3.org/TR/1999/REC-xpath-19991116</xacml3:
        XPathVersion>
  </xacml3:PolicyDefaults>
  <xacml3:Target>
    <xacml3:AnyOf>
      <xacml3:AllOf>
        <xacml3:Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <xacml3:AttributeValue
            DataType="http://www.w3.org/2001/XMLSchema#string">alice</xacml3:
                AttributeValue>
          <xacml3:AttributeDesignator
            AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
            DataType="http://www.w3.org/2001/XMLSchema#string"
            Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject"
            MustBePresent="false"
          />
        </xacml3:Match>
      </xacml3:AllOf>
    </xacml3:AnyOf>
  </xacml3:Target>
  <xacml3:Rule
      Effect="Permit"
      RuleId="http :// axiomatics.com/alfa/ identifier / alice . policy . Id_18">
    <xacml3:Description />
    <xacml3:Target />
  </xacml3:Rule>
</xacml3:Policy>
```

Figure 2.18: A permissive XACML policy.

SecPAL. Despite XACML having a mechanism for delegation its semantics, like the rest of the language, are poorly defined in natural language. The standard gives one example of how delegation in XACML can be used: a company has a printer and Carol is responsible for saying who can use the printer. She delegates to Bob who in turn grants Alice the right to use the printer. An excerpt from the XACML delegation policy example is shown in Figure 2.19. The full policy set is split into three smaller policies[1]

```
'company' says 'carol' can-say inf
  Person:X canUse(Printer:P).

'carol' says 'bob'' can-say
  Person:X canUse(Printer:P).

'bob' says 'alice' canUse(Printer:P).
```

Using ALFA we can also write delegation policies in the style of XACML. An example (taken from [15]) might be a policy for a bank where only the account holder, or the account holder's guardians can view their account:

```
policy account{
  target clause object.objectType == "account"
  apply firstApplicable
  rule viewAccount{
    target clause action.actionId == "view"
    condition (user.username == account.owner) ||
      (stringIsIn(stringOneAndOnly(user.username),owner.guardians))
  }
}
```

The equivalent in AppPAL would be:

```
'bank' says Person:O canView(Account:A)
if A hasOwner(O).

'bank' says Person:G canView(Account:A)
  if A hasOwner(O),
     O hasGuardian(G).

'bank' says Person:P can-say
  P hasGuardian(Person:G).
```

[1] An individual decision in XACML is called a *policy*. Many policies form a *policy set*.

```
<PolicySet PolicySetId="PolicySet1"
          Version="1.0"
          PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-combining-algorithm:permit-overrides"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
          xsi:schemaLocation="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17 xacml-core-v3-schema-wd-17.xsd">
  <Target/>
  <Policy PolicyId="Policy1"
         Version="1.0"
         RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides">
    <Target>
      <AnyOf> <AllOf>
          <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue
                DataType="http://www.w3.org/2001/XMLSchema#string"
                >employee</AttributeValue>
            <AttributeDesignator Category="urn:oasis:names:tc:xacml:3.0:attribute-category:delegated:urn:oasis:names:tc:xacml:1.0:subject-category
                   :access-subject "
                                 AttributeId="group"
                                 MustBePresent="false"
                                 DataType="http://www.w3.org/2001/XMLSchema#string"/>
          </Match>
        </AllOf> </AnyOf>
      <AnyOf> <AllOf>
          <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue
                DataType="http://www.w3.org/2001/XMLSchema#string">printer</AttributeValue>
            <AttributeDesignator Category="urn:oasis:names:tc:xacml:3.0:attribute-category:delegated:urn:oasis:names:tc:xacml:3.0:attribute-
                   category:resource"
                                 AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
                                 MustBePresent="false"
                                 DataType="http://www.w3.org/2001/XMLSchema#string"/>
          </Match>
        </AllOf> </AnyOf>
      <AnyOf> <AllOf>
          <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue
                DataType="http://www.w3.org/2001/XMLSchema#string">print</AttributeValue>
            <AttributeDesignator Category="urn:oasis:names:tc:xacml:3.0:attribute-category:delegated:urn:oasis:names:tc:xacml:3.0:attribute-
                   category:action"
                                 AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
                                 MustBePresent="false"
                                 DataType="http://www.w3.org/2001/XMLSchema#string"/>
          </Match>
        </AllOf> </AnyOf>
      <AnyOf> <AllOf>
          <Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue
                DataType="http://www.w3.org/2001/XMLSchema#string">Carol</AttributeValue>
            <AttributeDesignator
                Category="urn:oasis:names:tc:xacml:3.0:attribute-category:delegate"
                AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
                MustBePresent="false"
                DataType="http://www.w3.org/2001/XMLSchema#string"/>
          </Match>
        </AllOf> </AnyOf>
    </Target>
    <Rule RuleId="Rule1" Effect="Permit">
      <Target/>
    </Rule>
  </Policy>
```

Figure 2.19: Excerpt from XACML delegation example [87].

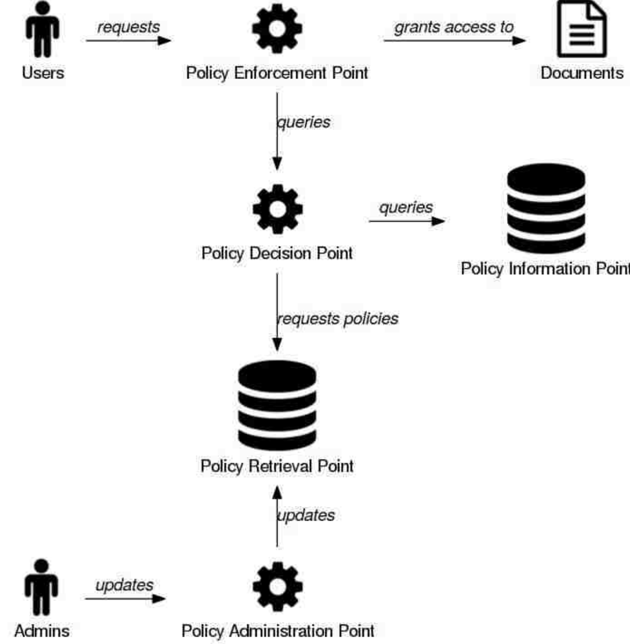


Figure 2.20: The XACML reference architecture.

XACML also defines a reference architecture for anyone using XACML, shown in Figure 2.20. The architecture consists of five major *policy points*. The reference architecture describes how to setup the policy points so that queries can be made and answered automatically. The PEP stands between the users and the files (for example) they are trying to access. When a user tries to access a file the PEP sends a query to the PDP (which makes the *decision*), and *enforces* the policy by either granting or denying access. In order to make the decision PDP *requests* policies from the PRP (which are updated by an *administrator* editing the policies at the PAP), and using *information* from the PIP.

XACML's architecture fits well with a centralised architecture, where distributed PEPs talk to local PDPs which retrieve policies from a centralised PRP. For a decentralised scenario the architecture fits less well as the decision point may have to retrieve policies and decisions from multiple sources (which may in turn require more decisions and policies to be retrieved).

SecPAL Extensions. We started this chapter, by describing SecPAL. Since SecPAL was first described various tweaks and extensions were made to the language. SecPAL was extended to allow existential queries. This allowed it to answer queries such as *"does Alice say Bob can read any of her files?"* or *"do*

all Alice's apps meet her installation policy?", which could only be answered by manually making multiple queries before. Becker also described the possibility of dynamic assertion retrieval, which would allow SecPAL to fetch and add assertions to its assertion context when making queries. In the case of delegation this would allow a delegated principals assertions to be imported dynamically rather than having to be present in the AC before evaluating a query. Becker defined a safety condition, but didn't describe a protocol for retrieving assertions, however [22].

Becker also noted that roles and the *can-act-as* statement had proven to be of limited use. Delegation using *can-say* could replace it with greater control in all cases Wherever that *can-act-as* to use exclusively depth bounded delegation (as we showed in Section 2.1.1).

2.3 Fine-Grained Permission Systems

Whilst policy languages, in general, give us a mechanism to describe trust relationships and decisions *in general*. Fine-grained permission systems are a type of policy language to control app behaviour on Android. In this section we survey some of the developments here.

This section looks exclusively at Android as that is where the bulk of the research has been. It is relevant to our work as these alternative permissions schemes have been used to describe the policies about how users wish to use their devices. The majority of the work has looked at Android: perhaps because, unlike iOS, the operating system can be modified to incorporate new features.

To access some APIs, such as access to private storage, Android apps must ask for a permission. Permissions are a coarse usage control mechanism. Until 2015 (and Android Marshmallow), users could not control which permissions an app had (by default) and not installing an app was the only way to stop an app accessing data, without modifying Android. Since Android Marshmallow, users can disable many of an app's permissions after installation.

The coarseness of Android's permission system, and its relative inflexibility, has led to a line of research into *fine-grained* permissions systems. These systems add extra permissions controls, and allow greater control of their enforcement. We describe several notable examples and what extra controls or new enforcement means the fine-grained permissions systems offered. A

fine-grained permissions system is designed to enforce app policies, whereas our work on AppPAL is about capturing the broader policies of the entire mobile ecosystem. A fine-grained permissions system could be *plugged into* AppPAL, via the constraints mechanisms, to allow enforcement of fine-grained permissions policies, if we wished to capture and use these tools.

Some of the earliest work on fine-grained permissions systems for Android starts with Enck et al.'s work on Kirin [45]. Kirin allowed a user to define what behaviour they considered acceptable using a policy language. Kirin ran on the device to certify apps using static analysis. If an app did not conform to the policy the user would be warned and could uninstall it. A natural next question is rather than rejecting the entire app, can you just reject the behaviour you don't like? Ontang et al.'s SAINT tool [91] did just that for inter-app communication, allowing the user to express a policy about the circumstances two apps could share data or call functionality. Similarly, Apex [85] allowed user's to specify at install time constraints as to when a permission could be used. CRePE [36] took these ideas further allowing permissions to be denied based on *context* (for example the user was on a train, or the user was within Bluetooth range of their computer).

Dr. Android and Mr. Hide [69] was a fine-grained permission scheme that worked by rewriting apps to use guarded APIs. AppGuard [17] worked similarly to Dr. Android, but allowed for controlled responses when a permission was denied—essentially ensuring that the app didn't crash if it didn't get access to the data it expected to. Aurasium [114] works by modifying apps to monitor system calls and IPC mechanisms via GOT hijacking in the Bionic libc, allowing their policies to effect native code which earlier tools ignored.

Jin et al. developed a fine-grained permission system that targeted apps written using the PhoneGap framework, that enable developers to create portable native apps from web apps [70]. Dai et al. proposed a fine-grained permissions scheme that controlled apps (and IoT devices) access to media stored on cloud based services based on a user's policies [38].

2.4 Moving Forward

This chapter described SecPAL, and a number of other policy languages designed in the same time frame. We also described a number of fine-grained permissions systems that have been used to enforce policies on Android. In the next chapter we show how we took SecPAL and instantiated it to describe the policies surrounding mobile devices. In doing so we created AppPAL. We describe the language and our implementation of it in Chapter 3.

AppPAL fits into this background of policy languages by instantiating SecPAL for a new domain. It does not have new semantic language features to describe new kinds of policies. Instead, its novelty lies in the application to a domain that has not before had its policies captured using a precise language.

Chapter 3

Instantiating and Evaluating SecPAL

The mobile ecosystem (which we define as the devices, users, stores, developers and policies surrounding the ways we use mobile devices) contains many interacting entities, including devices interacting with their users, app stores selling software, developers building apps, wireless access points devices connect to, and the companies and networks the devices and their users work within.

Every entity in the ecosystem has its own policies. For example, there is a contract between the store selling apps and the developer programming them. The stores have content policies as to which apps they will sell, often prohibiting sexual or illicit content. The app developers may have preferences as to which stores they will sell their apps in, favouring stores with higher market shares, or which take smaller cuts of the app sale price. Not all policies are as formal as a contract: a user may have preferences about which kinds of apps he wants to install but may not write these preferences down. Instead, he might pick apps to install from a store based on his best guess whether it matches his preferences. The mobile ecosystem is a *distributed system*. Phones, users and stores are not aware of each other until they interact—there is no list of every app store, every device and every user. They must make policy decisions on their own without relying on a central authority to make decisions for them. They can, if required, delegate to others for their policies and knowledge.

Formal languages let us write policies without ambiguity and describes precisely the process behind making decisions. A policy written in a formal lan-

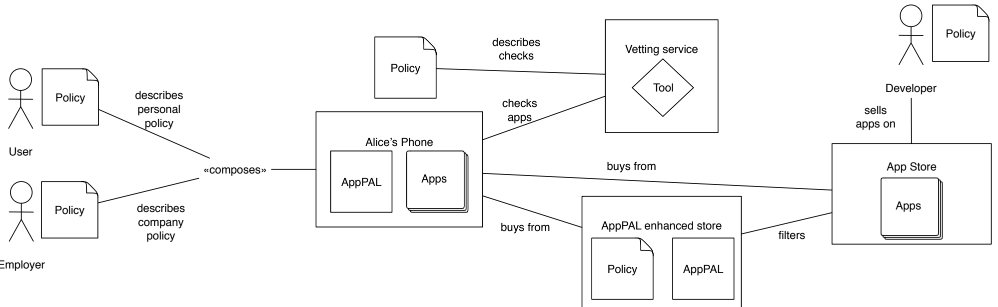


Figure 3.1: Entities in the mobile ecosystem and some of the policies surrounding them.

guage describes what decisions can be made. Translating policies into a formal language allows for rigorous comparisons of different policies, preferences and rules in mobile ecosystems. Writing the policy in a formal language gives a precise meaning and *perhaps* allows it to be checked automatically, if the policy is decidable.

Becker et al. designed SecPAL to make access control decisions in distributed systems [23]. SecPAL is expressive, has a clear and intuitive syntax, is decidable and extensible. That SecPAL is extensible is important as we can apply the language to a new domain by instantiating it with predicates and constraints.

This chapter introduces how SecPAL is instantiated to describe the policies surrounding mobile ecosystems. We explain why SecPAL is a good choice to describe policies in the mobile ecosystem. I introduce AppPAL as an instantiation and implementation of SecPAL for mobile device policies.

3.1 Why SecPAL

There are many policy languages. Some target specific domains: such as Ponder which targets firewalls, OS's and databases [39]. Others, such as Cassandra, focus on credential management [20]. Some are general like XACML [88]: designed to express access control decisions from a variety of domains. Different policy languages are described in greater detail in Chapter 2.

The mobile ecosystem has specific requirements for a policy language to describe its policies. Namely, we need a policy language that can express:

Delegation. Sometimes entities may wish to share information and policy rules. A user might delegate to an expert user to help them decide what policies they want to use. An app store might use information from an app vetting firm to decide what apps to sell. A company might want HR and IT departments to be responsible for some decisions, and they in turn might wish to give specific HR or IT workers specific decisions to make. Capturing these relationships lets us describe precisely who makes what decisions in the mobile ecosystem.

Locality. In the mobile ecosystem different devices can have different policies. For example, two users of mobile phones may disagree about what makes an app *installable* (i.e. acceptable to install). Equally, the policies a store may wish to enforce are very different from the policies a user may wish to enforce. The locality of a query is the place where the decision is being made, typically in the immediate vicinity of an entity such as a user or store. Each entity in the ecosystem also expects to enforce their own policies—there is no overarching enforcer of policies, rather users enforce whatever preferences they have, stores enforce their own rules through whatever means they see fit. Being able to point to the location a decision is made is important when considering delegation: a user might ask a friend if an app is good: we want to distinguish between the friend telling the user directly, and the friend giving the user sufficient information to draw the conclusion themselves.

Access external information. There are many tools and sources that can give us information about apps (as well as other subjects). These tools each give information in their own way: be it employee data through an SQL database, terminal output through a command line analysis tool, or HTML output onto an app store's web-page. We want to be able to make use of this information when writing policies, but we should not rely on all tools standardising on our policy language as a universal information transfer mechanism. We want our policy language to be able to capture the policies which use these external sources without forcing the tools

themselves to work in any particular manner. In other words, the policy *specification should be separate from its enforcement*.

Constraints. As well as external information, we'd also wish our policy language to capture dynamic information such as the time of day or the current location of a device. This information is sometimes incorporated into a policy language through *constraints*. This would allow us to start to create rules that depend on the context of the user. For example a company boss might want to check what time their employees get into work. To do this the boss requires their employees install an app that gives him their location. Employees might allow their boss to check on them during the work day, but during the weekend and the evenings the employee would rather their boss could not see where they were.

Flexibility of Policy Style. The mobile ecosystem encompasses many scenarios. A company looking to enforce a mobile device policy would use a MAC-style policy, but a user trying to filter what APIs an app can access may write a policy in a DAC-style. A user trying to describe might be happy to install an app with certain permissions if the app has a particular purpose. For example a user might be happy to install a file management app that can access a memory card, but not happy to install a torch app with the same set of permissions—a policy reminiscent of an RBAC or ABAC policy. We want to capture all these policies, and we would want our policy language to be agnostic to any particular policy style.

We chose SecPAL as the basis for our policy language for mobile ecosystems as it provides the features we wanted. By using its *can-say* and *can-act-as* phrases we can capture different delegation patters (as described in Section 2.1.1).

Every assertion in SecPAL is made by an explicit speaker, who should sign each statement with a key. This gives locality to statements. If we have an assertion from Alice that she thinks an app is good then it must have come directly from Alice. If we just have assertions from Alice that describe the general process for deciding if an app is good or not, then we might reasonably believe that the decision whether Alice would consider an individual app good or not might be made elsewhere.

SecPAL's constraint mechanism (the *where* part of an assertion) lets us implement the constraints we described but also allows us to access external

information. Constraint functions can be written to run a particular tool, or make a query to a database, allowing us access to external information.

SecPAL policies are style agnostic, and the language does not prescribe a particular style to write polices in. It is easy to capture idiomatic policy styles using SecPAL. For example, a classic example of a DAC-style policy are UNIX file permissions. The decision by an OS whether to allow a user (U) to read a file (F) based on the permissions on the file-system (FS) can be written in SecPAL:

```
'os' says User:U canRead(File:F)
  if F hasOwner(U),
     F hasPermissionsMask('400').

'os' says User:U canRead(File:F)
  if F hasGroup(G),
     U isMemberOf(G),
     F hasPermissionsMask('040').

'os' says User:U canRead(File:F)
if F hasPermissionsMask('004').

'os' says 'fs' can-say File:F hasOwner(User:U).
'os' says 'fs' can-say File:F hasPermissionsMask(Mask:M).
```

Similarly, we can also capture the *simple* and *star* MAC rules of the Bell-LaPadula model (the *read-down, write-up* rules) using SecPAL:

```
'admin' says User:U canRead(File:F)
  if U isSecurityLevel(LU),
     F isSecurityLevel(LF)
  where LU >= LF.

'admin' says User:U canWrite(File:F)
  if U isSecurityLevel(LU),
     F isSecurityLevel(LF)
  where LU <= LF.
```

Writing RBAC, or ABAC-style policies is similarly easy. Unlike some languages, such as Cassandra [20] or RT [78], which encourage a certain style of policy, SecPAL doesn't say *how* a policy should be written, it instead lets the policy author write their rules as they see best.

As well meeting our requirements for a policy language, we agree with Becker et al.'s assertion that SecPAL is readable and has evaluation rules that

are easy to understand and unsurprising [23]. This again makes SecPAL a good fit to describe the policies of the mobile ecosystem as even someone unfamiliar with SecPAL might be able to look at an assertion and be reasonably expected to understand what the assertion means.

Tschantz and Krishnamurthi defined various reasonability properties (properties that determine how well someone might predict how a policy will behave) for policy languages [108]. They note that compared to languages based on first-order logic (like SecPAL):

> "policies written in XACML are more transparent than policies written in languages based on first-order logic"

In their paper, Tschantz and Krishnamurthi compare a restricted version of XACML (Core XACML) to three first-order logic based languages: FOL, Lithium and $\mathcal{L}_5$. Tschantz and Krishnamurthi go on to define their properties: *totality* if the language always makes a decision, *determinism* if the language always makes the same decision given the same query, *monotonic* if the policy makes the same decisions if the rules are reordered, *safety* if adding a facts from a policy only allows more permit decisions to be made and *independent composition* if the results of querying the policy as a whole are no different from querying subsets of the policy's rules and combining them. They note that whilst Core XACML is deterministic it is not safe, does not have independent composition and is not monotonic.

SecPAL in contrast is total (by the closed world assumption), is monotonic (there are no policy combinators and negation is not allowed in policies), and has independent composition (there are no policy combinators). If the policy is always structured to use the same form of decision (the policy always uses `canRead` instead of `canNotRead`, for example) then the language is safe too. SecPAL without constraints is also deterministic, and if constraints are used then the language is deterministic only if the constraints themselves are[1]. SecPAL seems, therefore, to meet most of Tschantz and Krishnamurthi criteria for a reasonable language, despite being based of first-order logic.

Other languages, perhaps such as XACML or Ponder, could have been used as a starting point for a policy language. We chose SecPAL, however, as the basis for AppPAL—a policy language for mobile ecosystems.

[1] A constraint which rolled a dice and checked if the number rolled was 2 would not be deterministic but could be used in SecPAL.

3.2 Basic Examples of AppPAL

AppPAL is a policy language that describes the policies of the mobile ecosystem. To be more precise: it is SecPAL, with a slightly modified syntax to allow for additional typing constraints (described in Section 3.3.2) and to aid machine parsing. As well as the modified syntax, AppPAL also instantiates SecPAL (which is generic) to with the predicates and types to talk about mobile security policies (Section 3.3).

One example of the policies AppPAL might describe is a user selecting which apps they want to use on their phone. Suppose a user, Alice, decides that an individual app (Angry Birds, for example) is okay to use on her phone. This is written as:

```
'alice' says 'com.rovio.angrybirds' isInstallable.
```

Having decided the app is the one Alice wishes to use she tells the device's package manager that it must install the app. The package manager might be connected to an app store (such as the Play Store on Android: the `com.android.vending` APK file), a MDM program such as iOS's Volume Purchase Program (VPP), or an IT manager manually provisioning devices. The user may not know who fills the role of the *package manager* but most systems provide one nevertheless.

```
'alice' says 'package-manager'
  mustInstall ('com.rovio.angrybirds').
'alice' says 'com.android.vending' can-act-as 'package-manager'.
```

Additionally Alice may allow her workplace to dictate some apps that must be installed (a delegation).

```
'workplace' says 'alice' mustInstall('com.microsoft.office.word').
'alice' says 'workplace' can-say
  'alice' mustInstall('com.microsoft.office.word').
```

This policy works with single apps: Alice decides which apps to install and lists them. This is an accurate description of how users currently interact with stores and their devices. Users may have more complicated rules for deciding what to install, but following the rules is usually left to the user's own self-discipline.

Policy languages allow for greater generality. A cautious user may only

install Android apps with certain permissions[2].

```
'user' says App isInstallable
  if App hasntPermission('CAMERA'),
     App hasntPermission('INTERNET'),
```

Others may avoid apps that allow them to spend money within the app.

```
'user' says App isInstallable
  if App cannotMakeInAppPurchases.
```

Some users may rely on an AV program installed on their phone. The phone can only install apps checked by the AV program.

```
'user' says App:A isInstallable
  where runAV(A) = 'safe'.
```

Alternatively a user may allow an app to be installed if two or more of their friends are willing to recommend the app to them.

```
'user' says App isInstallable
  if App isRecommendedBy(Friend1),
     App isRecommendedBy(Friend2)
  where Friend1 != Friend2.

'user' says Friend:F can-say
  App:A isRecommendedBy(F).
```

3.2.1 AppPAL Policies for App Stores

AppPAL gives a language for describing these decision-making processes. By writing down policies in formal language we not only start to allow machine-based decision-making, avoiding the need for user's self-control, but we also can compare different policies. This goes beyond describing user's app preferences. Consider the two largest mobile operating systems: Apple's iOS and Google's Android. When comparing the systems Apple's is often described as a *walled-garden* whereas Android is an *open-platform* [18, 44]. Language like this is informal. It hints at the differences without giving them precisely.

Using AppPAL we can model the differences between the two systems and make more meaningful comparisons. The *walled-garden* comments relate to

[2]Permissions are the access control mechanism for device features used by Android.

their different app store models. Users of iOS can only install apps from the *App Store*[3]. If a user is willing to install a special certificate, from a developer or business, however they may install apps through the browser or a computer connected to the device. Apple controls these special certificates, issuing and revoking them. They must also be authorised by the device's owner.

```
'device' says App isInstallable
  if App hasBeenSignedBy(Cert),
     Cert hasIssuer('apple').

'device' says App isInstallable
  if App hasBeenSignedBy(Cert),
     Cert hasIssuer(X),
     'apple' hasAuthorized(X),
     'device' hasOwner(U),
     U hasAuthorized(Cert)
  where validCertificate(App, Cert) = true.

'device' says Authority:A can-say A hasAuthorized(Certificate:C).
```

In contrast, an Android user is free to install any signed app (though who signed it is not immediately important). Unless the user has enabled *side-loading*, the app must come from the Play Store, as opposed to a file downloaded through another app store or the internet. Alternatively, if the user has enabled *developer-mode* then they may install apps through the Android Debug Bridge (ADB).

```
'device' says App isInstallable
  if App hasSource('play-store'),
     App hasBeenSignedBy(Cert)
  where validCertificate(App, Cert) = true.

'device' says App isInstallable
  if App hasSource('file')
     'device' hasOwner(U),
     U hasAuthorized('sideloading'),
     App hasBeenSignedBy(Cert)
  where validCertificate(App, Cert) = true.
```

[3]*The App Store* is Apple's marketplace for selling apps, an *app store* is a generic term for an on-device store selling apps.

```
'device' says 'settings-app' can-say
  U hasAuthorized('sideloading').

'device' says App isInstallable
  if App hasSource('adb')
     'device' hasOwner(U),
     U hasAuthorized('developer-mode').

'device' says 'settings-app' can-say
  U hasAuthorized('developer-mode').

'settings-app' says 'user' can-say
  'user' hasAuthorized(Setting:X).
```

This glosses over some details of certificate handling (though the example could be extended further). It also ignores how the system might update apps. It does, however, provide a far more precise version of the differences in app installation between the two platforms. For iOS, to install an app Apple have to authorise it, either by signing the app or the issuer. In contrast, for Android, to the device can install any app if the device owner is willing to enable the relevant setting.

3.2.2 Worked Example of Policy Checking

As a worked example consider a user Alice, trying to install an app on her work phone. Alice works for a hospital trust and the trust has a mobile device policy. The trust's policy states that if Alice wants to install an app Alice has to clear it with her immediate manager. If it is for clinical use then Alice also has to clear it with the Care and Clinical Policies Group (CACPG). If it is for business use then Alice also has to clear it with the Management of Information Group (MIG). Finally, the Integrated Governance Comitee (IGC) should clear the app. This is a relatively complicated policy but realistic: it comes from an actual NHS trust's BYOD rules (shown with an AppPAL translation in Appendix A).

To implement the policy the trust publishes the following AppPAL rules:

```
'nhs-trust' says App isUsable
  if App hasMet('clinical-use-case').

'nhs-trust' says App isUsable
  if App hasMet('business-use-case').

'nhs-trust' says 'cacpg' can-say
  App:A hasMet('clinical-use-case').

'nhs-trust' says 'mig' can-say
  App:A hasMet('business-use-case').

'nhs-trust' says App isInstallable
  if App hasMet('final-app-approval'), App isUsable.

'nhs-trust' says 'igc' can-say
  App hasMet('final-app-approval').

'nhs-trust' says Device canInstall(App)
  if App isInstallable, App isApprovedFor(Device).

'nhs-trust' says Employee:Manager can-say
  App:A isApprovedFor(Device)
  if Manager isResponsibleFor(Device).
```

Suppose Alice wishes to install the app `ms.office` for business purposes. To satisfy the policy and install the app she needs to collect the following statements.

- `'nhs-trust' says 'ms.office' isInstallable`. For this, she needs the MIG to state that it has a business use-case. She also needs approval from the IGC.

  1. `'mig' says 'ms.office' hasMet('business-use-case').`
  2. `'igc' says 'ms.office' hasMet('final-app-approval').`

- `'nhs-trust' says 'ms.office' isApprovedFor('alices-device').`

  To get this she needs a statement from the manager responsible for Alice's device (suppose it is *Bob*) approving the app.

  3. `'bob' says 'ms.office' isApprovedFor('alices-device').`

4. `'nhs-trust'` *says* `'bob' isResponsibleFor('alices-device').`

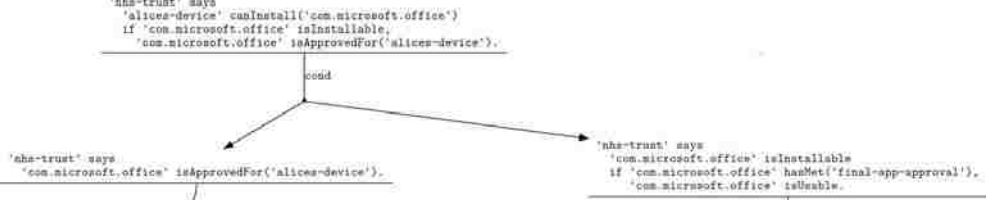
- Additionally, she needs the following typing statements.

5. `'nhs-trust'` *says* `'ms.office' isApp.`

6. `'nhs-trust'` *says* `'bob' isEmployee.`

Alice obtains the statements by contacting the speakers. Each may either give her the statement she needs or may give her more rules in the form of a cryptographically signed electronic representation of each assertion. For the MIG and IGC may authorise the app or they might give a list of additional checks they need before they agree to authorise the app. When checking if the app is an App in Item 5, the NHS trust might delegate further. They could reply that if the App is in the Google Play store then they recognise it as a valid app. Alice would then have to get more assertions if she wanted to prove this statement. Alternatively, the speaker could refuse to make the assertion, either because they do not believe it, or they cannot give an answer. In this case, Alice would have to look for an alternative means to prove the assertion or accept that they cannot install the app.

Once Alice has collected the statements, either by contacting the speakers directly or through other means, Alice can use a SecPAL inference tool, such as AppPAL, to check whether she can install the app. Additional to the decision, if a proof is found, a tool like AppPAL can output a proof tree[4] to show how it made the decision (Figure 3.2). The proof tree shows the AppPAL assertions used to prove each statement at varying levels of the tree: from the top-level goal which, in this case, used the *cond* rule first shown in Figure 2.8.

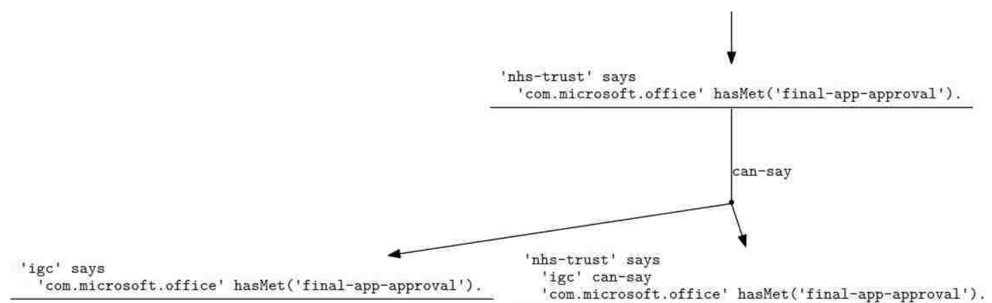


To later nodes which might use SecPAL's other rules to be proved (*can-say* in this case).

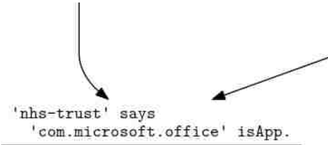
[4]Technically a DAG.

Prefix	Meaning
subject **can***Action*	The subject is allowed to perform the action.
subject **has***Action*	The subject has performed the action.
subject **is***Property*	The property holds true for the subject.
subject **must***Obligation*	The subject is required to satisfy the obligation.

Table 3.1: Standard prefixes used for AppPAL predicates.



If two nodes depend on the same sub-goals, then the proof will be reused.



3.3 Instantiating SecPAL for Mobile Ecosystems

SecPAL is a generic language. In SecPAL's grammar (Figure 2.7), predicates and constraint functions describe the decisions and checks done in a particular domain. The choice of predicates and constraints defines the decisions the SecPAL instantiation can talk about.

AppPAL instantiates SecPAL to describe policies in mobile ecosystems. It was initially focused on describing app installation policies, however it was later extended further to describe other policies, such as BYOD policies.

3.3.1 Predicate Conventions

When instantiating SecPAL we use predicates based on four verbs: *can*, *has*, *is* and *must*. These verbs describe facts common to many policies, such as permitted actions, completed actions, describing properties of entities and obligations respectively.

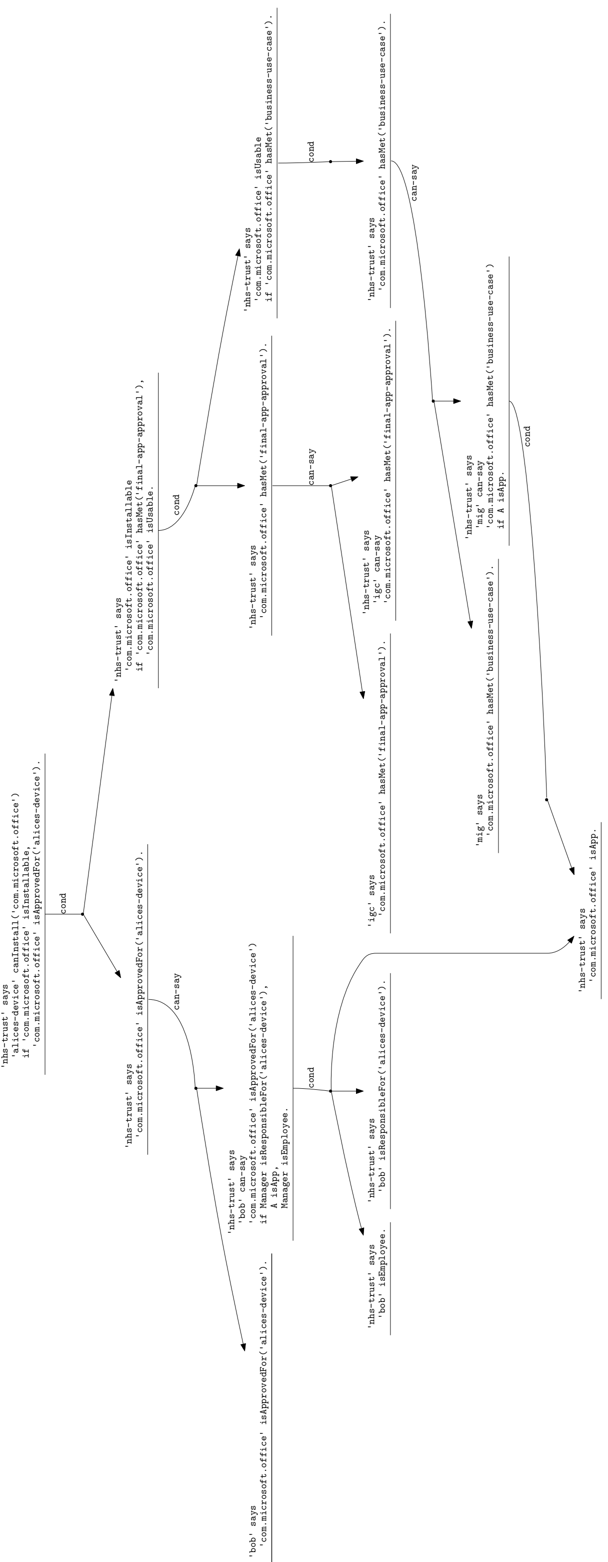


Figure 3.2: Proof tree generated by AppPAL when checking whether to install Alice's app.

subject *is*Property A statement that says the *subject* has a given *property*. A common example of this is to restrict variables to a given type. An example might be that `'angry-birds' isApp` or that `'jennie' isEmployee`. Constants and variables in SecPAL are untyped, so it is often helpful to restrict their type to a given domain. For example, Alice might trust Bob to describe which apps are good, but not what websites.

```
'alice' says 'bob' can-say X isGood if X isApp.
```

Additionally, SecPAL requires that all variables in the head of a statement be used in the body (the restriction helps the language to be reducible to Datalog^{C}, and thus ensure all queries terminate): assigning a type by using an *is* statement is sufficient to satisfy the rule.

Not all constants or variables need have a property or type, but some may have multiple ones, for example an app might have the *good* property, the *app* type, and be *installable*. Each of these properties is expressed using the *is* statement. There is no obligation to provide a type to variables or constants, but when writing policies we have found them helpful as they help specify precisely the domain a rule should operate on.

Is predicates are sometimes used as goals for queries, for example to test whether an app *isInstallable* or not. They also often occur frequently inside assertion bodies, as described.

subject *has*Action Describes actions the *subject* has completed.

For example if an app on a device has requested a permission then we might write:

```
'device' says App:A hasRequestedPermission(Permission:P).
```

If a device requires its owner to grant a permission we might write:

```
'device' says User:U can-say
  App:A hasBeenGrantedPermission(Permission:P)
  if 'device' isOwnedBy(U).
```

Often, has predicates fall into common patterns and groups, for example *hasRequested* and *hasBeenGranted* here. There is no requirement for the predicates to be named in such a way—*hasBeenGranted* could be named

canUseFeaturesAssociatedWith. In practice when describing policies it is best to be consistent and use just one predicate for one action or property to avoid confusion, redundant rules, and multiple ways of satisfying a policy. In Section 3.5 an early implementation of a tool for checking for duplicated predicates is described.

Has predicates are rarely used as top-level queries as they describe actions already taken. They might be used as a goal if a policy needs to describe actions occurring in a given order. For example, a company might require that devices report if a user is using a company device unacceptably:

```
'company' says User:U hasBehavedUnacceptably
  if U hasAccessedWebpage(W),
    W isPornographic.
```

subject *canAction* An authorization. The *subject* may perform the *action*. *Can* predicates are used to describe access control decisions. For example, a company might wish to limit access to a server to employees who working in R&D. To do so they might write a policy such as:

```
'company' says User:U canConnectTo('192.168.20.22')
  if U isInDepartment('r&d').
```

Other examples of policies involving *can* predicates might include a user describing whether an app is allowed access to their photographs, or an app store deciding whether to allow a developer to sell their app in the store.

Can predicates are frequently used as goals in queries. They can be used to describe access control decisions, and as such are often the source of a query whether a subject *can do* some action.

There is no obligation on someone who uses a policy that has *can* predicates, to also generate an assertion using a *has* predicate when an action is performed. For example, an app store may have terms and conditions that any user can read.

```
'app-store' say User:U canReadTermsAndConditions.
```

A user, Alice, might read these terms and conditions having satisfied the rule that she can do so. Having done so it might be reasonable to expect

the store to add to its knowledge base an assertion that:

```
'app-store' says 'alice' hasReadTermsAndConditions.
```

Adding assertions in this manner is not required, however, though keeping these relationships can be helpful. In Section 6.2 we discuss some potential ways we could use these relationships for auditing and policy compression, if the relationships were to be enforced.

subject *must*Action An obligation. The *subject* should carry out the *action*. An example might be requiring the device tell a company's IT department if there have been three unsuccessful password attempts:

```
'company' says Device:D mustInform('it', 'login-failure')
  if D hasUnsuccesfulLogins(N)
  where N >= 3.
```

AppPAL does not state how quickly a subject should complete their obligation, or enforce that they even perform it. These additional checks could be created by using additional predicates:

```
'mum' says 'alice' hasSatisfiedObligationToCall
  if 'alice' mustCall,
     'alice' hasCalled.
```

Must predicates are mostly goals, as they describe the conditions when its subject must do something. There are exceptions, such as the example above where *must* predicate is a condition, but they are unusual. If a policy were to contain a particular *must* predicate as a condition, but never as a goal, then the policy would be unsatisfiable—something we can check for as described in Section 3.5.1.

Some of the predicates may seem to allow AppPAL access to information from outside of its assertion context. In order to use external information, however, an AppPAL constraint must be used. For example, if we had a fact: `User:X canWriteTo(File:f)`, then we might imagine it implemented with a rule:

```
'admin' says User:X canWriteTo(File:f)
  where fsAllowWrite(X, F) = true.
```

Where `fsAllowWrite()` is a constraint that checks whether the file-system would allow a given user access to file. An alternative to using a constraint, would be

$$\langle E \rangle ::= \langle \text{Variable} \rangle \mid \textbf{'constant'}$$
$$\langle \text{Variable} \rangle := \textbf{Type:Var} \mid \textbf{Var}$$

Figure 3.3: Changes to SecPAL's syntax to support types. Changed terms are shown in red.

to generate ground AppPAL assertions before running the query which state exactly which users can write to any given file.

```
'admin' says 'alice' canWriteTo('alices-documents').
'admin' says 'alice' canWriteTo('project-notes').
'admin' says 'bob' canWriteTo('project-notes').
```

When there are large numbers of users and files this may become infeasible however.

Our predicate conventions differ in the approach to the ones added with the SecPAL4P and SecPAL4DSA languages [24, 16]. Both these languages add extra phrases to SecPAL's grammar. For example SecPAL4P adds a *may* and *will* phrase to describe whether to carry out an action. If Alice, a policy author, wished to use SecPAL4P to say that someone can forget her email address she could write:[5]

```
Alice says x may delete Email within t
```

The same AppPAL rule would be:

```
'alice' says User:X canDeleteWithin('email', Time:t)
  where currentTime() < t.
```

3.3.2 Type Notation

When writing a policy, it is common to use conditions in facts that limit the scope of a variable. To do this we use *is*-predicates, that give their subject a type. For example *Alice* might declare that *Bob* is responsible for saying which apps she can install. This can be written in SecPAL as follows:

```
'alice' says 'bob' can-say App isInstallable
  if App isApp.
```

[5] SecPAL4P also relaxes safety rules that permit the variables x and t to be in the head of the rule, but not the body.

AppPAL

```
X says T1:A predicate(T2:B, T3:C)
  if ... .
```

SecPAL

```
X says A predicate(B, C)
  if A isT1,
     B isT2,
     C isT3,
     ... .
```

Figure 3.4: De-sugaring from AppPAL types to SecPAL.

```
Expand Types(Assertion):
    for v ∈ Assertion.Head() do
        if ∃ v.Type() :
            f ← new Fact(v, "is"+v.Type())
            Assertion.Body().Add(f)
    return Assertion
```

Figure 3.5: Procedure to expand types from AppPAL into SecPAL.

When writing this we added a condition `if App isApp`, that Bob can only talk about Apps as being installable. Generalising this pattern we use predicates starting with *is* to give types to their subjects. If a policy rule contains a lot of variables, however these typing conditions can become very verbose. To simplify the policy rules, AppPAL adds a sugared notation for typing statements by extending SecPAL's grammar for variables (Figure 3.3).

Expansion of the AppPAL types into SecPAL conditions is described in Figure 3.5. This is run when parsing AppPAL code, and adds a rule to AppPAL that if a variable in the head of an assertion has a type then it is removed and a condition that the variable is that type is added to the body of the assertion. If a variable in the body of an assertion has a type, then AppPAL's parser reports it as an error when reading the policy. If a variable needs multiple types, they can be added by adding additional *if* predicates to the body.

Using this sugared notation, the earlier example becomes:

```
'alice' says 'bob' can-say App:A isInstallable
```

For a more complex example consider the following example taken from a BYOD policy.

```
'company' says Device:D canConnectToAP(AP:X)
  if X isOwnedByCompany.
```

The rule states that the company will only allow devices to connect to company owned access points. The syntactic sugar expands into the following equivalent policy.

```
'company' says Device canConnectToAP(X)
  if X isOwnedByCompany,
     Device isDevice,
     X isAP.
```

This is a fairly simple refinement of SecPAL's syntax, but it improves the readability. It avoids hiding the condition that the company must own the access point among typing statements aiding readability. It also helps avoid errors in policies caused by the policy author forgetting to give a type to a variable.

The typing syntax can also lead to scenarios where the typing *is* predicates are checked multiple times when querying a policy. For example, in the following hypothetical policy if Alice queried whether *Angry Birds* was installable she might check three (or more) times if `'alice' says 'angry-birds' isApp.`

```
'alice' says App:A isInstallable
  if A isAGame.

'alice' says App:A isAGame
  if A hasCategory('game').

'alice' says 'google-play' can-say App:A hasCategory('game').
```

In practice, however, this isn't an issue as our implementation (described in Section 3.4) caches the results from queries when evaluating policies. We also describe in Section 3.5.2 a method for detecting when conditions are redundant.

We also make no attempt to detect when types are clashing or inconsistent. For example, we might expect that an object could be an app, or a device but not both. If a policy were to contain, for example assertions that:

```
'alice' says 'angry-birds' isApp.
'alice' says 'angry-birds' isDevice.
```

We might speculate the policy may contain errors as we wouldn't expect Angry Birds to be both an app and a device. AppPAL does not by itself check for such errors, however, if desired a policy author could add an assertion to their policy to check for such type-errors.

```
'alice' says 'policy' hasTypeErrors
  if X isApp,
     X isDevice.
```

To our knowledge this is the first attempt to add types to SecPAL. Adding a static type-checker, and looking at more complex type systems has been left to future work.

3.4 Implementation

To our knowledge no prior open source implementation of SecPAL exists. Becker implemented SecPAL as a closed-source library atop the .net framework [97]. This library included examples and a C# API for SecPAL, allowing assertions to be created using code, but no parser for SecPAL. Since Becker's SecPAL implementation cannot be trivially extended and will only run on Microsoft Windows, the decision was made to re-implement AppPAL from scratch as an open source library.

Our AppPAL implementation is a Java library, with roughly 5,000 lines of code. The implementation is available online[6]. The library creates an AppPAL instance with an empty cache. This instance can be given several policies to enforce. The architecture is shown in Figure 3.6. The instance is queried and will give decisions based on whether the queried assertion is valid according to the policy. As part of the constraint checking AppPAL can also be connected to external databases, systems, and static analysis tools. These can give AppPAL with more information external to that provided by SecPAL at the time of checking, and connect AppPAL to tools to enforce the policies.

Implementing AppPAL as a library allows us to embed it into a variety of situations. AppPAL can be part of an app store checking the apps sold against a policy. Running on a device, AppPAL can check apps before installation by the package manager. There is also a command-line version which is useful for

[6] `https://github.com/apppal/libapppal`

testing and modelling policies. Our implementation can make the inferences, on an Android device or on a command line, but it is not integrated into Android. It is future work to evaluate how AppPAL policies might work in practice and be tied into a mobile operating system to enforce policies in practice. The work here looks primarily and modelling the policies and preference and providing a framework for enforcement, rather than the enforcement itself.

The AppPAL interpreter implements SecPAL's evaluation rules (shown in Figure 2.8) directly. This differs from Becker's original description of SecPAL [23] which describes evaluation through Datalog^C. Datalog^C is a variant of Datalog extended to support constraints [76].

Becker used a translation from SecPAL to Datalog^C in order to prove certain properties about SecPAL: namely decidability and tractability (with polynomial data complexity in the size of the policy). To implement SecPAL efficiently he described a novel Datalog evaluation algorithm using tabling, as the *bottom up* methods were inefficient with a changing assertion context, and the *top down* methods (such as SLD resolution) could run into infinite loops when evaluating *can-say* or *can-act-as* statements. To solve these issues Becker proposed an algorithm that extended SLD resolution with a table to prune infinite search trees by tracking which rules had been used before.

When implementing SecPAL we considered using a Datalog^C back-end but could not find an implementation. We also explored using a Datalog implementation (such as Z3 or Datomic) to build SecPAL but in practice no implementation could fully support Datalog^C's constraints. To implement the constraint checking we would need to modify the Datalog implementation to run additional checks whilst making inferences. For AppPAL these additional checks might require running an external program, but this would not be a trivial change to make for the libraries we looked at.

We also wanted our implementation to run on an Android device (we were interested in having users enforce policies on their phone), and all the Datalog libraries we found could not be ported trivially.

Instead of implementing Datalog^C ourselves, AppPAL implements SecPAL's evaluation rules directly and adds Becker's tabling method to avoid infinite loops. Our implementation is naïve and not the optimal method for evaluating queries. Faster solutions might use answer-set programming and an SMT solver to answer queries. For all policies we tried, our implementation was fast

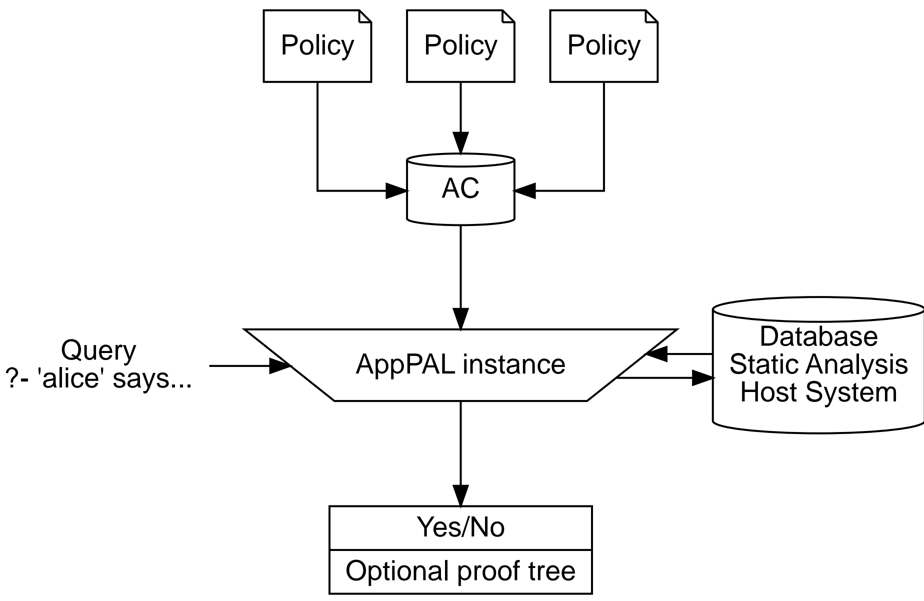


Figure 3.6: AppPAL's inputs and outputs.

enough—answering all but the most complex queries in seconds (Section 3.4.3).

3.4.1 Evaluation

Pseudo-code for the evaluation algorithm used by AppPAL is described in Figures 3.7, 3.9 and 3.8.

To make queries against a policy, AppPAL is first given one or more policy files. AppPAL parses the files and adds the assertions within them to the assertion context (AC). The AC is then preprocessed to extract more information; this is summarised in Table 3.2. If a constant is used before a *says* statement, or it is used as the subject of a *can-say* fact then the constant is marked as *voiced*. A constant is *voiced* if it is the speaker of a statement or it is the subject of a *can-say* fact. A constant is a *subject* is it is the subject of a fact. The predicates used in the policy are also extracted and marked as *derivable*. This allows some queries to be decided automatically and some rules to be flagged as unusable in the assertion context. This lets AppPAL reject some queries quickly: if a query has a speaker that isn't voiced then we cannot query what the speaker says. If a rule relies on un-derivable predicates, then it cannot be used to make decisions.

$c \in$ Voiced	$\Longleftarrow$	$\exists\,(c$ says $\cdots) \in$ AC	$\bigvee$
		$\exists\,(\star$ says c can-say $\cdots) \in$ AC	
$c \in$ Subjects	$\Longleftarrow$	$\exists\,(\star$ says $c \cdots) \in$ AC	$\bigvee$
		$\exists\,(\cdots$ if $\cdots, c\ \star, \cdots) \in$ AC,	
		$c \in$ Constants	
$p \in$ Derivable	$\Longleftarrow$	$\exists\,(\cdots \star\ p(\cdots) \cdots) \in$ AC	

Table 3.2: Sets used in AppPAL evaluation.

A results table (*RT*) is also created. All ground facts in the assertion context are added to the results table as proven facts. The table stores partial results and previously established proofs. The table is indexed by queries and a delegation depth, partial results (i.e. that the evaluation of this query is ongoing). This allows previous results to reused without re-computation or constraint re-evaluation. It also prevents AppPAL proofs growing unboundedly (and the decision process not terminating). If when searching for a proof we meet a query that we are currently evaluating, i.e. one that exists higher in the current proof tree, we treat it as false. Multiple queries will share the same results table until the table is cleared, or the AppPAL instance is stopped.

Ignoring queries we are currently evaluating lets us avoid proofs with cycles (i.e. proof trees where the proof of a query depends on proving itself). If we can find a terminating proof with a cycle then we can find also shorter proof by removing the cycle. If a proof tree has a cycle that never terminates (i.e. it loops infinitely) then we can treat it as effectively false as it would take an infinite amount of time to derive.

When evaluating the policy we also track which rules and predicates we have used. From this we can reconstruct a proof tree that shows how AppPAL made a decision. This allows an auditor to check the decision-making process and aids AppPAL's developers with debugging.

AppPAL's evaluation procedure is not the same as SecPAL's, though in practice both could be used to evaluate AppPAL or SecPAL policies. SecPAL was evaluated by first converting queries and policies into Datalog$^{\mathsf{C}}$. Becker then described a novel Datalog$^{\mathsf{C}}$ evaluation algorithm with tabling to evaluate the converted queries and make decisions. The AppPAL evaluation algorithm

```
Evaluate(AC, RT, Q, D):
    if RT.Contains((Q, D)) :
        result ← RT.Get((Q, D))
        if result=Success :
            return result
        else:
            return Failure
    else:
        RT.Set((Q, D), (Inprogress, p))
        p ← Cond(AC, RT, Q, D)
        if p=Proven :
            RT.Set((Q, D), Success)
            return Success
        else:
            p ← CanSay-CanActAs(AC, RT, Q, D)
            if p=Proven :
                RT.Set((Q, D), Success)
                return Success
            else:
                RT.Set((Q, D), Failure)
                return Failure
```

Figure 3.7: Pseudo-code for evaluating a query.

```
Cond(AC, RT, Q, D):
    for a ∈ Assertions(AC) do
        u ← Q.Unify(a.Head())
        if u.IsValid?() :
            a ← a.Apply(u)
            for θ ∈ PossibleVariableSubstitutions(AC, a) do
                a′ ← a.Apply(θ)
                if Variables (a′) = ∅ :
                    if ∀ b ∈ a′.Body(): Evaluate(AC, RT, b, D)=Success :
                        if CheckConstraint(a′.constraint)=True :
                            return Success
    return Failure
```

Figure 3.8: Pseudo-code for using the cond-rule.

```
CanSay-CanActAs(AC, RT, Q, D):
    for c ∈ Constants(AC) do
        if c ∈ Subjects(AC) :
            if CanActAs(AC, RT, Q, D, c) = Proven :
                return Proven
        if D = ∞ ∧ c ∈ Speakers(AC) :
            if CanSay(AC, RT, Q, D) = Proven :
                return Proven
    return Failure

CanActAs(AC, RT, Q, D, c):
    q1 ← Q.speaker says c can-act-as Q.subject.
    q2 ← Q.speaker says c Q.verbphrase.
    p1 ← Evaluate(AC, RT, q1, D) = Proven
    p2 ← Evaluate(AC, RT, q2, D) = Proven
    if p1 ∧ p2 :
        return Proven
    else:
        return Failure

CanSay(AC, RT, Q, D, c):
    q1∞ ← Q.speaker says c can-say inf Q.fact.
    q2∞ ← c says Q.fact.
    p1∞ ← Evaluate(AC, RT, q1, ∞) = Proven
    p2∞ ← Evaluate(AC, RT, q2, ∞) = Proven
    q1 ← Q.speaker says c can-say Q.fact.
    q2 ← c says Q.fact.
    p1 ← Evaluate(AC, RT, q1, 0) = Proven
    p2 ← Evaluate(AC, RT, q2, 0) = Proven
    if (p1∞ ∧ p2∞) ∨ (p1 ∧ p2) :
        return Proven
    else:
        return Failure
```

Figure 3.9: Pseudo-code for using the can-say and can-act-as rules.

does not make use of Datalog^{C}, but instead implements AppPAL's evaluation rules directly. We do, however, borrow some strategies from Becker's SecPAL evaluation algorithm—AppPAL's results table serves a similar purpose to SecPAL's tabling caching results and preventing cycles. The use of sets to restrict the search space for variables is unique to the AppPAL algorithm, however.

Our method for evaluating AppPAL offers an alternative to Becker's procedure. We developed it to avoid having to implement Datalog^{C}, and to allow us greater control over how AppPAL evaluated policies. We believe the procedure is equivalent to Becker's SecPAL evaluation algorithm (though a more rigorous comparison as to how the procedures compare has not been done).

3.4.2 Soundness and Completeness of Decision Procedure

The algorithm as described is neither sound nor complete as it makes use of a results table *RT*, but has no means to invalidate results in it. This causes problems when handling constraints, in particular temporal ones.

Consider a constraint that states that the current time must be between 9 AM and 3 PM. At 8:59 AM we try to evaluate a query that depends on this constraint being true (i.e. all possible proof trees use this constraint). No proof can be found so the `Evaluate` function correctly returns failure and the *RT* is updated accordingly. At 9:01 AM the query is re-run. Since a past result already exists in *RT* it is reused and evaluate returns failure again, despite the constraint now being satisfiable. Therefore, the algorithm is incomplete as it has failed to find a proof when one exists. A similar argument for soundness can be made by evaluating a query successfully at 2:59 PM and then again at 3:01 PM. A successful result may be returned which is unsound.

We use the results table to cache results, and to avoid circular proof searches. It helps makes the algorithm fast, albeit at the cost of soundness and completeness. If AppPAL were incorporated into a production system, the designers would need to have a strategy to manage the *RT* and delete old results. A simple strategy, such as clearing any statements older than 10 minutes, may be enough for most scenarios. Alternatively the cache can be cleared between queries, though this loses some of the speed benefits of having some common sub-query results cached. Ultimately the decision of how to manage the results table is up to the developer.

The arguments in the rest of the section assume that the result of evaluating a constraint does not change over time. This allows us to argue the correctness of the algorithm, assuming that the *RT* cache is correct.

When describing SecPAL Becker also showed that decision procedure he used for SecPAL was both sound and complete. His decision procedure had to parts: a translation from SecPAL into Datalog^{C}, and a Datalog evaluation algorithm involving tabling. He proved the soundness and completeness of both parts in Appendix C of the SecPAL technical report [23]. To do so, he described the SecPAL evaluation algorithm as a labelled transition system and proved by induction which states were reachable from an initial state. In comparison to Becker's proofs, the arguments presented here for AppPAL work through induction on the height of the proof tree generated by applying AppPAL's evaluation rules. Becker did not have to consider constraints when presenting his proofs as the SecPAL evaluation algorithm does not cache SecPAL queries. It does make use of a results table, but this stores the Datalog translation of the SecPAL statement without constraint only, and so the problems associated with the results of constraints changing over time do not apply.

Soundness Argument

$$\texttt{Evaluate}(AC, RT, Q, D) = \mathit{Success} \implies AC, D \models Q.$$

Base Case. For the base case we argue that the `Evaluate` function is sound when not making recursive calls. A query is `Evaluate`-d at a given delegation depth, against a given *AC* and *RT*. If the query has been made before, then the result will be present in *RT*, and will be reused. Therefore, in showing `Evaluate` is sound we must show that only queries that can be proven are ever marked as successful in *RT*.

The `Evaluate` function calls `Cond` function which implements SecPAL's *cond* rule. It searches the AC for unifying assertions (i.e. those assertions which have a head that is equal to the query under a variable renaming), then searches for possible variable substitutions that remove all variables. If the body is missing (i.e. the unifying assertion has no conditionals) and the constraint is true, then `Cond` will return success, and the results table will be updated accordingly. This corresponds to SecPAL's *cond* rule (we have greyed out the recursive case), hence we would argue it is sound as there is no other means for our evaluation

algorithm to mark a result as true in RT without recursively calling `Evaluate`. We call the call tree of `Evaluate` $T^1_{\mathtt{E}}$.

The analogous proof tree to $T^1_{\mathtt{E}}$ (called $T^1_{\models}$) produced by the evaluation rules, also relies on the *cond* rule (shown below with recursive elements greyed out). Both $T^1_{\mathtt{E}}$ and $T^1_{\models}$ have height 1.

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

Inductive Hypothesis. Suppose we know that we know that if `Evaluate(AC, {}, Q, D)` implies that $AC, D \models Q$, where the height of the call tree of `Evaluate` is n (in calls of `Evaluate`), and that the height of the implied derivation tree is also n. We hypothesise that if `Evaluate` is sound with a call tree and derivation tree height of n, then when `Evaluate`'s call tree height is $n+1$, there will be an equivalent derivation tree with height $n+1$.

Inductive Step. Let $T^n_{\mathtt{E}}$ be the call tree from a call to `Evaluate` with corresponding proof tree $T^n_{\models}$, both with height n. Let $T^{n+1}_{\mathtt{E}}$ be a call tree of height $n+1$ where $T^n_{\mathtt{E}}$ is a child of the root node. To show the hypothesis is true we must show that however the `Evaluate` function proceeds, it must correspond to one of AppPAL's evaluation rules; so the corresponding proof tree $T^{n+1}_{\models}$ is the matching evaluation rule with $T^n_{\models}$ as (one of) its children.

We can ignore the results table (if it starts empty, or only contains sound results) as it is just memoising results which could be recalculated without memoisation. We do not need to consider to the case of non-terminating queries (i.e. circular), as `Evaluate` will return `Failure` when the loop is detected.

`Evaluate` will call `Cond`, `CanSay` or `CanActAs` each which implements the corresponding evaluation rule and each of which calls `Evaluate`, where at least one call of which will produce $T^n_{\mathtt{E}}$. The corresponding proof tree will have the equivalent evaluation rule at its root and $T^n_{\models}$ as its child. Therefore, `Evaluate`$(AC, RT, Q, D) = Success$ implies we can always find an equivalent $AC, D \models Q$.

Completeness Argument

$$AC, D \models Q. \implies \texttt{Evaluate}(AC, RT, Q, D) = \mathit{Success}$$

Base Case. Consider the case where a query has been evaluated with AppPAL's evaluation rules and found true. Suppose the height of proof tree used to derive it is 1, that is to say that we have only used the *cond* rule once to show the query is valid. In this case the query must be unifiable with a fact in the AC (i.e. one with no *if* part). If this is the case, when using the `Evaluate` function we will call `Cond`, which will search the AC for assertions to unify with. Since (at least) one assertion must exist (without an *if*) for the *cond* rule to return true, then we will find it and `Evaluate` will always also return `Success`, without recursively calling itself.

Induction Hypothesis. Suppose know that for a proof tree of a query q (given an AC, and D) with height n, when running `Evaluate(AC, {}, Q, D)` it will eventually return `Success` (with at least n recursive calls to `Evaluate`). We hypothesise that for a proof tree of height $n+1$, `Evaluate` will also return success (given the same AC and q), with a recursive call depth of n.

Inductive Step. The `Evaluate` function will (in the worst case) search the entire AC for a rule which will unify with the currently evaluated query, and evaluate to `Success`. We argue that evaluate implements each of the AppPAL evaluation rules by comparison between the code and the inference rules.

For a proof tree of height $n + 1$, where $n > 0$, we will use one of the *cond*, *can-say* or *can-act-as* rules to derive the proof tree at height n. Each of the `Cond`, `CanSay`, and `CanActAs` functions which implement these evaluation rules will call `Evaluate` at least once (corresponding to the dependent proofs in the proof tree). The induction hypothesis says that these will be complete and will have at most n recursive calls to *Evaluate*. So at height $n + 1$ the algorithm must be complete as if the proof tree is valid, `Evaluate` would be able to evaluate the corresponding assertions and with a call tree height of $n + 1$.

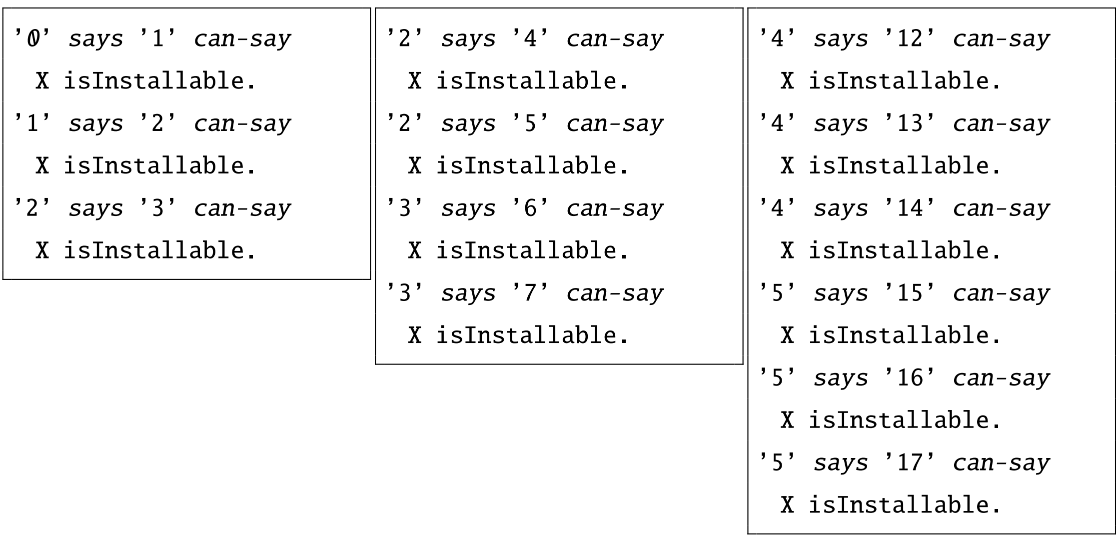


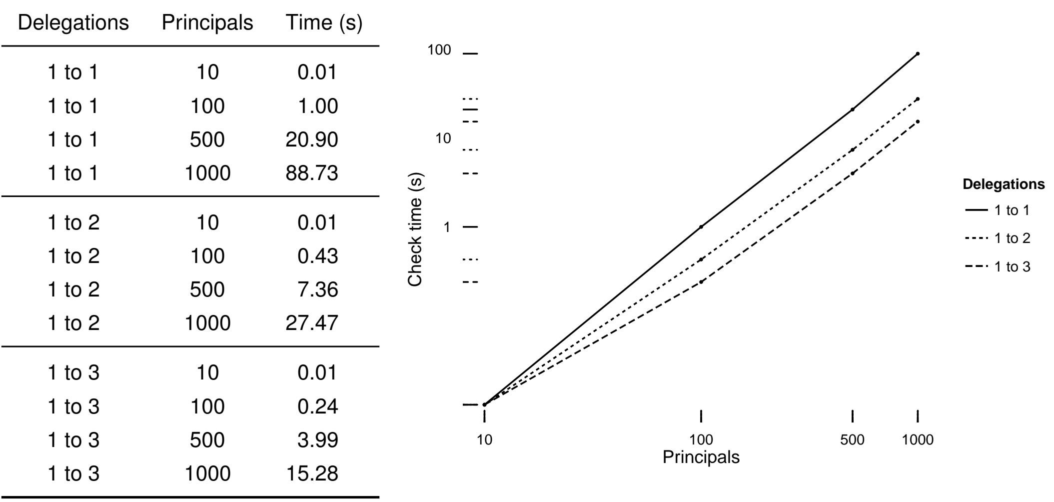
Figure 3.10: Excerpts from the 1 to 1, 1 to 2 and 1 to 3 benchmarks.

3.4.3 Benchmarks

Someone who uses AppPAL might wish it to check apps before installation. 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.

A synthetic benchmark is used to give a measure AppPAL's performance. The policy checking procedure is at its slowest when having to delegate repeatedly; the depth of the delegation tree is the biggest reason for slowing the search. Synthetic benchmarks checks 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 test—though worse ones could be designed by delegating even more or triggering an expensive constraint check.

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 Table 3.3. Most typical policies (such as those discussed in Chapter 4 and Chapter 5), use only few delegations per decision. We believe the policy



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

Table 3.3: Benchmarking results on a Nexus 4 Android phone.

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, even on a power constrained device such as a mobile phone.

3.5 Automatic Analysis of AppPAL policies

When examining an AppPAL policy it is natural to wonder whether the policy is as optimal, in terms of the rules and facts required to decide a query and the number of rules in the policy, as it could be. Is a decision reachable given the rules and facts contained in the policy? Does an assertion context contain enough statements to use a given rule? If there are multiple ways of deciding whether some statement is true or not, does one rule require fewer statements than any other? Does one rule require only a subset of the facts of another rule, implying the second is redundant?

Policies can be checked for many properties including:

Satisfiability. Do there exist rules which can never be satisfied?

Equivalence. Do there exist rules which have the require the same ground facts to be proven in order to satisfy them?

Redundancy. Do there exist multiple rules to make a decision where one is satisfied by a subset of the facts that satisfy the other?

Consistency. Can two contradictory decisions be made?

We have implemented preliminary checkers for two of these properties. The code is included with AppPAL. The output shown in `teletype` is output by the tools we have produced, and the diagrams shown in Section 3.5.2 are produced when running the tools in debug mode. Additionally, a checker for *consistency* could be implemented by using the same technique as the Ponder authorization language and disallowing predicates of the form can⋆ and cannot⋆ [39].

3.5.1 Checking Satisfiability

An assertion is *satisfiable* if there are sufficient facts such that the assertion's conditions can be met. This is related to the concept of *satisfiability* in SMT solving (where a propositional logic formula is satisfiable if there is an assignment of truth values to the propositional variables that makes the formula true), but made more complex by the recursive nature of AppPAL's (and other policy and database languages) assertions. Instead of just looking for an assignment of variables, we must look to see if there are rules and facts sufficient that we can derive the satisfiable statement.

We care about this property when writing policies as it means that our assertions affect something. If an assertion is *unsatisfiable*, then this may indicate that we have failed to specify one of the conditions it depends upon.

In Datalog (and, more generally, in logic programming) a Datalog predicate, found by satisfying a rule, is satisfiable if there are sufficient ground facts in the database to deduce it. Formally, satisfiability in Datalog is defined in terms of the Intentional Database (IDB) (the Datalog *program*, analogous to a policy file loaded into an AppPAL assertion context), and the Extensional Database (EDB) (a set of facts from both the IDB and those which can be derived using rules). Satisfiability for Datalog is defined as [75]:

> **Satisfiability:** An IDB predicate s of a program P is *satisfiable* if there is some EDB D, such that P defines a non-empty relationship for s.

In AppPAL, when an assertion is satisfiable, there is a combination of facts that could satisfy its conditionals. If there are no facts that could ever satisfy the rule, then the policy may have a bug. An analogy can be drawn to an *conditional always false bug* in a conventional programming language: we have code (or in this case a rule) that can never be used as the conditions for using it can never occur.

Drawing from the Datalog definition of satisfiability, we define an AppPAL statement G as satisfiable if there exists some AC such that there is an assertion A within the AC and some assignment of variables θ such that $A\theta$ derives G, and that all the conditions in the body of $A\theta$ are also satisfiable. Formally, we define this as:

$$\begin{aligned} G \in \text{Satisfiable } \textit{if } \exists A \in \text{AC} :\ \exists\theta :\ G \equiv \text{head}(A\theta) & \\ \wedge\ (\text{conditionals}(A) = \emptyset & \\ \vee\ \forall G' \in \text{conditionals}(A\theta).\ G' \in \text{Satisfiable.}) & \end{aligned}$$

Our satisfiability checker works by examining which assertions each principal could possibly say from a given assertion context. When checking the satisfiability we ask whether there is a mechanism in the policy for the principal to use a given predicate. We use the following rules with a relaxed notion of $\in$, that allows variable unification:

$$\frac{\begin{array}{c}\exists A \in AC\ s.t.\ A \equiv (\texttt{principal says} \star \texttt{predicate if} \star p_1,\ \cdots,\ \star\ p_n\texttt{.}) \\ \forall p \in (p_i \cdots p_n) :\ AC \vdash (\texttt{principal}, p) \in \text{Satisfiable}\end{array}}{AC \vdash (\texttt{principal}, \texttt{predicate}) \in \text{Satisfiable}} \quad (1)$$

$$\frac{\begin{array}{c}\exists A \in AC\ s.t.\ A \equiv (\texttt{principal says} \star S\ \texttt{can-say predicate if} \star p_1,\ \cdots,\ \star\ p_n\texttt{.}) \\ \forall p \in (p_i \cdots p_n) :\ AC \vdash (\texttt{principal}, p) \in \text{Satisfiable} \\ AC \vdash (S, \texttt{predicate}) \in \text{Satisfiable.}\end{array}}{AC \vdash (\texttt{principal}, \texttt{predicate}) \in \text{Satisfiable}} \quad (2)$$

Informally the rules say when a policy (in an assertion context) supports the notion that a principal can say a predicate (that (`principal`, `predicate`) $\in$ Satisfiable). Rule (1) says that a principal can make statements using a predicate if the assertion context contains an example of them doing so; and that if that example has conditionals, they policy also supports them saying each conditional's predicate. Rule (2) handles the case with delegation: a principal can use a predicate (that (`principal`, `predicate`) $\in$ Satisfiable), if they say someone else can say the predicate, and the policy supports the other person being able to use that predicate (that (S, `predicate`) $\in$ Satisfiable)).

To give an example, if we have an assertion context with the following assertions:

```
'alice' says 'bob' recommends('angry-birds').

'alice' says 'claire' canInstall(App)
  if 'bob' recommends(App).

'alice' says 'claire' mustInstall(App)
  if 'bob' highlyRecommends(App).
```

The first assertion can be used to show that:

$$(\texttt{'alice'},\texttt{recommends('angry-birds')}) \in \text{Satisfiable}$$

The assertion is present in the policy and there are no conditions, so:

$$(\texttt{'alice'},\texttt{recommends('angry-birds')}) \in \text{Satisfiable}$$

The second assertion shows that:

$$(\texttt{'alice'},\texttt{canInstall(App)}) \in \text{Satisfiable}$$

In the second case we have a condition, but since the first assertion showed that:

$$(\texttt{'alice'},\texttt{recommends('angry-birds')}) \in \text{Satisfiable}$$

Then we allow (with the relaxed notion of $\in$) that:

$$(\texttt{'alice'},\texttt{recommends(App)}) \in \text{Satisfiable}$$

Finally the third assertion cannot be used to show that:

$$(\texttt{'alice'},\texttt{mustInstall(App)}) \in \text{Satisfiable}$$

As we have no way to show that:

$$(\texttt{'alice'},\texttt{highlyRecommends(App)})$$

We also add a somewhat weaker notion of satisfiability (Satisfiable*) which distinguishes statements that might be satisfiable, but depend on delegation, and where the delegated party has made no assertions about this predicate. This distinguishes the case where we have missing information from the case where a speaker has made an assertion that is itself unsatisfiable.

$$\frac{\begin{array}{c} AC \vdash (\texttt{principal}, \texttt{predicate}) \notin \text{Satisfiable} \\ \exists A \in AC \; s.t. \; A \equiv (\texttt{principal says} \star \texttt{predicate if} \star p_1, \; \cdots, \; \star \; p_n\texttt{.}) \\ \forall p \in (p_i \cdots p_n) : \; AC \vdash (\texttt{principal}, p) \in \text{Satisfiable} \vee (\texttt{principal}, p) \in \text{Satisfiable}^\star \\ \exists p \in (p_i \cdots p_n) : \; AC \vdash (\texttt{principal}, p) \in \text{Satisfiable}^\star \wedge (\texttt{principal}, p) \notin \text{Satisfiable} \end{array}}{AC \vdash (\texttt{principal}, \texttt{predicate}) \in \text{Satisfiable}^*}$$

$$\frac{\begin{array}{c} \exists A \in AC \; s.t. \; A \equiv (\texttt{principal says} \star S \; \texttt{can-say predicate if} \star p_1, \; \cdots, \; \star \; p_n\texttt{.}) \\ \forall p \in (p_i \cdots p_n) : \; AC \vdash (\texttt{principal}, p) \in \text{Satisfiable} \\ \nexists A \in AC \; s.t. \; A \equiv (S \; \texttt{says} \star \texttt{predicate} \cdots\texttt{.}) \end{array}}{AC \vdash (\texttt{principal}, \texttt{predicate}) \in \text{Satisfiable}^\star}$$

The satisfiability rules are reminiscent of AppPAL's *cond* and *can-say* rules shown in Figure 2.8, but far more general. AppPAL's evaluation rules decide if a specific assertion by a speaker is supported by an assertion context. In contrast, the satisfiability rules look at whether it is possible that the evaluation rules *could* decide that the assertion is supported—that the AppPAL evaluation rules could conceivably be satisfied, not that the evaluation rules are satisfied for a specific assertion.

This definition of satisfiability is not complete (it ignores relationships formed using *can-act-as*, as well as delegations to principals specified as variables). It is, however, useful as a debugging tool as it can quickly check that a policy contains enough statements to make any decision.

For a real example of the satisfiability checker in use, consider the following snippet taken from the NHS policy described in Chapter 5. The rule described in the policy is that an app must be approved by the IGC as well as by either the CACPG as well as the MIG depending on whether it is for clinical or business use. We describe this in AppPAL as:

```
'nhs-trust' says App isInstallable
  if App isApproved, App isUsableClinically.
'nhs-trust' says App isInstallable
  if App isApproved, App isUsableNonClinically.
'nhs-trust' says 'igc' can-say App:isApproved.
'nhs-trust' says 'cacpg' can-say App:A isUsableClinically.
'nhs-trust' says 'mig' can-say App:A isUsableNonClinically.
```

What apps in practice are approved for use? As the policy document notes, none of the groups or committees have ever approved an app in practice. When

running the satisfiability checker on this policy, it reports that (among other information) no app is installable.

```
$ java -jar Lint.jar --satisfiability example.policy
[INFO]: loaded 1/1 files of 6 assertions
Issues identified when checking satisfiability.
The following decisions may be unsatisfiable by their speakers:
'nhs-trust' says * isUsableClinically
'nhs-trust' says * isInstallable
'nhs-trust' says * isApproved
'nhs-trust' says * isUsableNonClinically

In particular the following assertions are unsatisfiable:
'nhs-trust' says App isInstallable if App isApproved, App
    isUsableNonClinically.
'nhs-trust' says App isInstallable if App isApproved, App isUsableClinically.

These decisions may be satisfiable through delegation but we
lack any statements to that effect from the delegated party:
(via 'cacpg') 'nhs-trust' says * isUsableClinically
(via 'igc') 'nhs-trust' says * isApproved
(via 'mig') 'nhs-trust' says * isUsableNonClinically
```

As well as reporting which decisions it cannot make, it also reports the specific assertions as well. It also reports the assertions that may be satisfiable through delegation given additional statements (the Satisfiable*) separately at the bottom.

These checks are simple and we don't take into account dependencies between variables. If we add, for example, the statements:

```
'igc' says 'angry-birds' isApproved.
'cacpg' says 'dropbox' isUsableClinically.
'mig' says 'instagram' isUsableNonClinically.
```

Then we will still never find any installable apps, as the IGC, CACPG and MIG need to agree on the same app to find it installable. Our rules for determining what is satisfiable only concern themselves with the predicates and the principals: no checks are done to ensure that the subjects of the policies, match up. In part this is because AppPAL has a closed-world assumption: if a principal does not talk about a specific constant with respect to a predicate then it is assumed false. When we run the satisfiability checker, we find no problems

as all the decisions are now satisfiable as there is a decision about *some* variable; even if that variable isn't useful in practice.

```
$ java -jar Lint.jar --satisfiability example.policy
[I]: loaded 1/1 files of 11 assertions
[I]: no satisfiability problems
```

The satisfiability checker acts as a quick sanity checker that a policy contains enough facts and assertions; unlike AppPAL-proper which can check how and whether a specific statement would be made.

3.5.2 Checking Redundancy

If unsatisfiability can be caused when we lack sufficient facts and assertions to make a decision then redundancy occurs when we have too many. Specifically there are two types of redundancy [74] that we care about here:

- *Unreachability* occurs if a predicate does not take part in the minimal derivation tree of a fact.
- *Irrelevance* occurs if a derivation tree contains pairs of identical atoms.

These ideas can be applied to AppPAL to detect some potential problems in AppPAL policies. For instance, consider the following AppPAL policy:

```
'alice' says App isInstallable
  if App isRecommended,
     App isNotMalware.

'alice' says App isRecommended
  if App isNotMalware,
     App isGood.
```

Alice checks that the app is not malware when checking the app is installable and when checking that the app is recommended. The check in `isInstallable` is irrelevant as it depends on the app being recommended which also checks this property. When writing AppPAL policies this kind of irrelevance commonly occurs when using the typed-syntax described in Section 3.3.2. For example, in this excerpt from a SANS BYOD policy there is a check that *U* is a user in both assertions. In the first there is irrelevance because *U* is stated as being a user twice, where once would have been sufficient. In the second there is a single

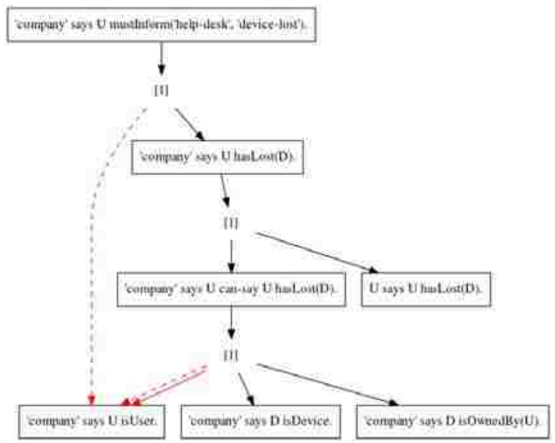


Figure 3.11: Proof graph showing irrelevance.

check that *U* is a user but it is irrelevant as the check had already been done when checking if *U* had lost the device (since only users can lose devices in the first assertion).

```
'company' says User:U can-say User:U hasLost(Device:D)
  if D isOwnedBy(U).

'company' says User:U mustInform('help-desk', 'device-lost')
  if U hasLost(Device).
```

Whilst the irrelevance adds redundancy and can slow down inferences when making policy checks, it can also aid policy comprehension. In the previous example the repeated checks as to whether someone is a user do not, strictly, need to be done. They do, however, clarify what the type of each variable is. A future implementation of AppPAL might incorporate code to remove the redundant checks automatically. In order to remove the redundant checks we must first be able to identify them, however.

We can check for irrelevance by building a proof-graph for the assertion context. Every node (shown in a box) represents an AppPAL fact we might wish to prove. For every assertion in the context we create a proof (represented as a number in brackets) for the head of the assertion, which is connected to the facts required to prove the assertion. In the case of can-say and can-act-as statements we expand them as per AppPAL's inference rules. A proof-graph for the above example is shown in Figure 3.11: the irrelevant links are shown in red and the AppPAL facts connected to the two dashed ones can be removed to remove the irrelevance.

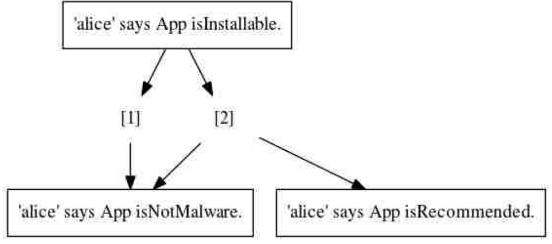


Figure 3.12: Proof graph showing unreachability.

In contrast, unreachability occurs when a fact does not take part in the minimal derivation tree of a fact. As a simple example consider the following policy:

```
'alice' says App isInstallable if App isNotMalware.
'alice' says App isInstallable if App isNotMalware, App isRecommended.
```

To detect unreachability for a given policy we again build the proof-graph (shown in Figure 3.12). For each proof node we collect the facts (the leaves of the graph underneath it, which are the ground assertions from the AC[7] in AppPAL). If the facts for one proof node connected to a fact are a subset of the facts for another proof node, then the larger proof node is unreachable as it contains facts which are not in the minimal derivation tree. In the case of Figure 3.12, the derivation-graph 2 is made redundant by derivation-graph 1 as it contains a subset of the facts. This is a simplified example; in general facts lower in the tree may have multiple derivation trees, leading to multiple sets of facts being required for a fact that seems to have only one proof node. Loops can also occur (where one fact depends on itself to prove itself). This approach isn't complete, but it does identify several cases where an AppPAL policy may be redundant through unreachability.

Redundancy can also occur when there are multiple rules that result in the same decision being made. Rules may depend on other rules, or ground facts. One proof (A) is made redundant by another proof (B) if the set of ground facts used in B is a subset of the ground facts used in A. Whenever A is satisfied B will also be, but when B is satisfied A may not be: consequently A is redundant

[7]Or in the case of an unsatisfiable policy facts with variables that cannot be unified with a ground fact.

```
'x' says 'y' p
  if 'y' q,
     'y' r.

'x' says 'y' p
  if 'y' q.
```

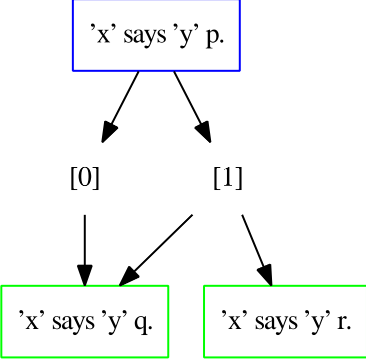


Figure 3.13: A simple policy shown as a graph.

as B can be used to prove its goal with fewer facts. Informally, for any goal G:

$$\exists p_1 \in \text{proofs}(G).\ \exists p_2 \in \text{proofs}(G).$$
$$p_1 \neq p_2 \ \wedge\ \text{facts}(p_1) \subset \text{facts}(p_2) \implies G \text{ has unreachable proofs.}$$
$$p_1 \neq p_2 \ \wedge\ \text{facts}(p_1) = \text{facts}(p_2) \implies G \text{ has equivalent proofs.}$$

Where proofs(G) is the set of all possible proof trees for a goal G.

Additionally, if two different goals (G and G') have equivalent proofs, i.e. there are multiple proofs that both rely on the same ground facts, then we report this as it implies the two statements may not be independent.

$$\exists p_1 \in \text{proofs}(G).\ \exists p_2 \in \text{proofs}(G').$$
$$\text{facts}(p_1) = \text{facts}(p_2) \implies G \text{ and } G' \text{ have equivalent proofs.}$$

A simple example might be the policy shown in Figure 3.13. The second rule makes the first redundant. We can represent the policy as a graph shown opposite the policy. The goal (shown as a blue rectangle) has two routes to prove it true (each shown in ellipses). Route 1 requires that the facts (shown in green rectangles) `'x' says 'y' r`, and `'x' says 'y' q.`, whereas route 0 only requires the latter fact.

A more complex example is shown below:

```
'x' says 'z' p if 'z' q.
'x' says 'y' can-say 'z' p.
'y' says 'z' p if 'z' q.
'y' says 'x' can-say 'z' q.
```

Representing this policy as the graph in Figure 3.14 we can see it is more complex. Goals that depend on more than just green facts, are shown as black

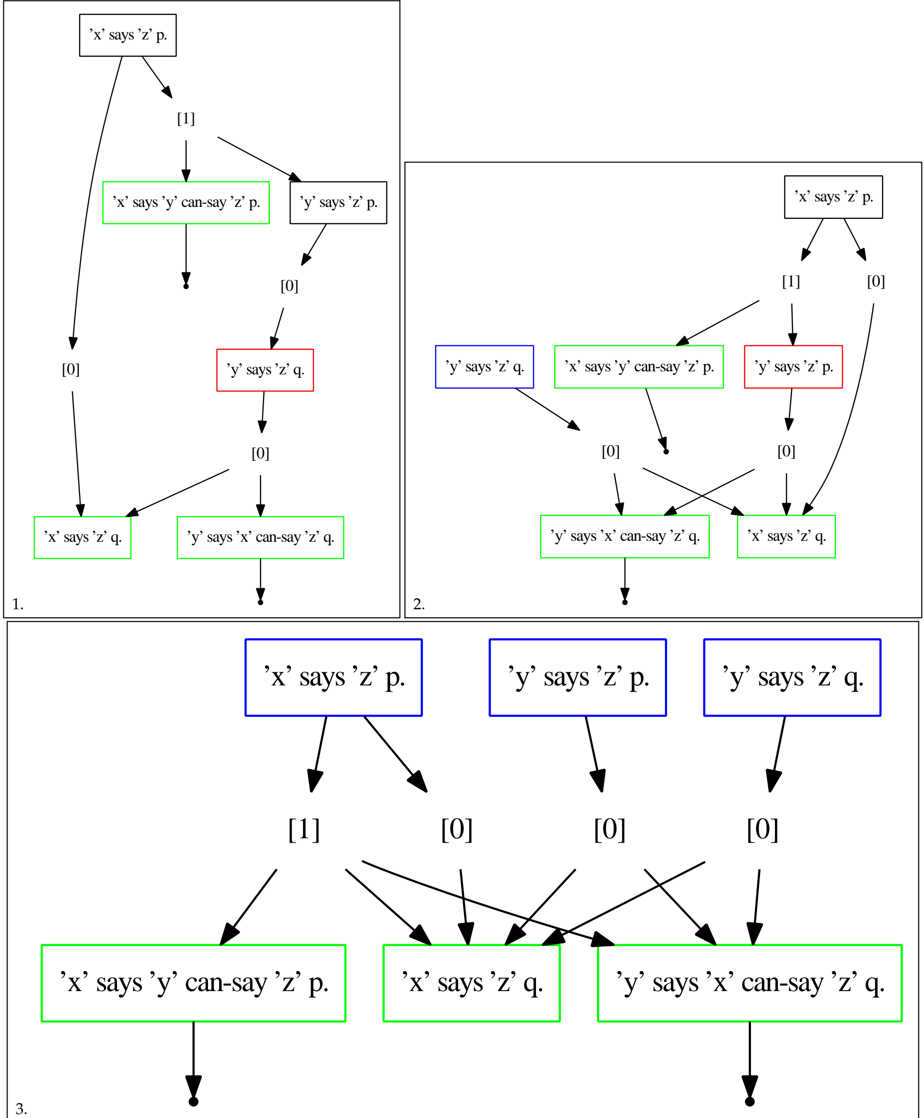


Figure 3.14: Flattening a more complex policy.

rectangles. If a goal is used to prove another goal, and it itself only depends on green, ground, facts, then the node is marked to be flattened (red rectangle). Its proofs are merged into the higher proof, and the flattened goal is removed from the higher proof. This process is repeated until no more nodes can be flattened (shown twice in Figure 3.14). Once the graph is flattened we can identify that `'x' says 'z' p` has a redundant means of proof (route 0 only uses one of route 1's facts). We can also see that all the proofs for `'y' says 'z' q` and `'y' says 'z' p` use the same facts. We report these statements as having equivalent proofs as the goals are not independent of each other (implying we could use fewer goals and still write equivalent policies).

Chapter 4

App Stores and App Preferences

Apps and app stores are a key part of the mobile ecosystem, an app being the key form of software and the app store as their primary distribution means. In this chapter we explore the stores and the policies surrounding them. We compare the terms and conditions between the various marketplaces. We look at how users have privacy preferences about the which apps they want to use. But by using AppPAL versions of common user privacy preferences, we show how Android users do not seem to follow them in practice. Finally, we describe how AppPAL can be used to create new app stores by checking for apps that can be shown to follow a policy, and offering them in a web app-based store.

4.1 App Stores

Whilst iOS has just one store (Apple's App Store), the Android ecosystem is more diverse with multiple stores available for multiple different purposes. The dominant app store on Android is Google Play. Unlike iOS, Google isn't the sole app vendor however. Some device vendors add their own stores to their devices as a feature: Amazon does this with their Kindle Fire tablets, where Kindle owners can download discounted apps. Some vendors, such as Samsung, add their own store this to highlight apps that use features specific to their phones (KNOX in Samsung's case). In some regions (such as China), using Google services is problematic due to legislation banning their use in the region. People in these countries use local, regionally focused app stores (such as QiHoo360 or Yandex) instead.

For a manufacturer to install Google's store they must comply with the

Android Compatibility Definition Document (CDD) [50]. The CDD describes how to configure the Android operating system and what features a device must have to run Android. If a manufacturer cannot pass the Compatibility Test Suite (CTS) that tests conformance with the CDD, then they cannot install the Play Store.

For manufacturers like *Jolla* whose devices do not run Android[1], but can emulate some parts of the Dalvik virtual machine enough to run Android apps, third-party stores like Yandex and Aptoide allow the device to still benefit from apps designed for the Android ecosystem, without having to pass Google's test suites.

In this section we will focus on five different app stores: Apple's App Store, Amazon's app store, Aptoide, Google Play and Yandex. Apple's App Store is for iOS, but the rest are for Android operating systems.

Amazon's app store was opened in 2011 and was the primary app store for Amazon's Android-based Kindle tablets[2]. Like Apple and Google's stores, the store is controlled by the device manufacturer.

Yandex is a store primarily for the Russian and eastern European markets. It features many Russian language apps. Some manufacturers, such as Nokia, installed Yandex over Google's app store on devices sold in Russia.

Aptoide is an open-source app store, and software for creating new app stores. Unlike many other app stores it provides specialised app stores targeted for Android based Smart TVs (Aptoide TV), and for users with slow internet connections (Aptoide Lite). Aptoide doesn't provide access to fixed sets of apps. Any user, or developer, is free to create their own *curated* repository of apps, and point Aptoide to it. Aptoide claim users and developers have created over 220,000 stores using Aptoide. Notably the F-Droid app store which sells only open source apps was created from a fork of Aptoide's code. Aptoide tries to detect malware in its own repositories (Figure 4.1), like many other store vendors. Unlike others rather than removing the malware Aptoide alert their customers that the app may be dangerous. Not all apps are subject to these checks however. Searching the store it was quickly possible to find an app designed to root phones (Figure 4.2)

We chose these stores to focus on as they represent a range of different app

[1]They run Sailfish, which is based off of the Linux Foundation's Mer operating system.
[2]The Kindle Fire models, as opposed to the regular Kindle models.

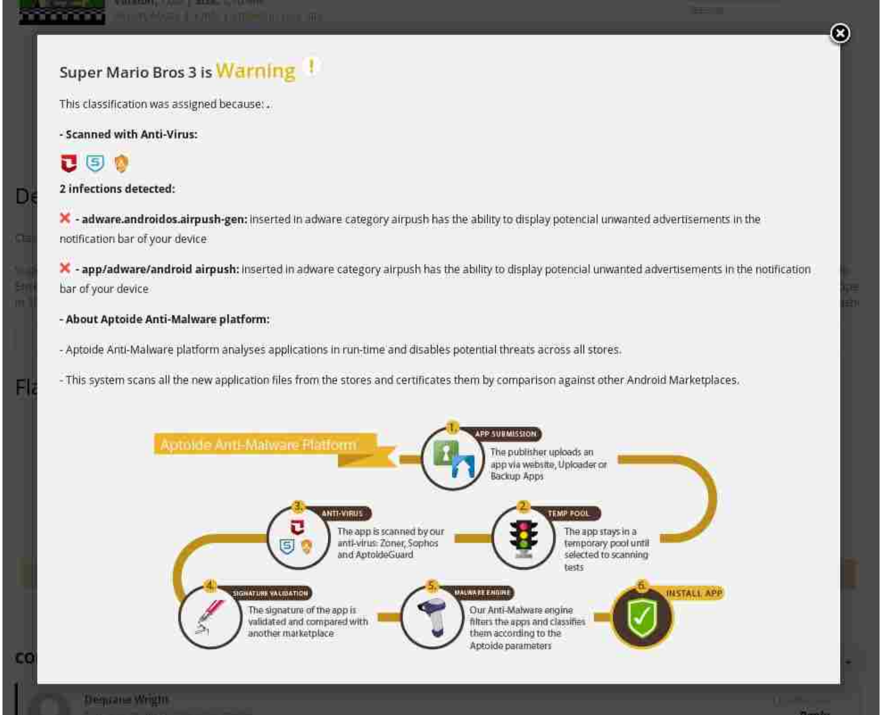


Figure 4.1: Adware infested and pirated app from Aptoide.

Figure 4.2: Rooting app found on Aptoide.

stores. From the largest of marketplaces (Apple and Google) to smaller markets for specific devices (Amazon) specific regions (Yandex) and for companies and users looking to create their own stores for their own devices (Aptoide).

4.1.1 Exploring Differences in Terms Between App Stores

With the various app stores available, a user or developer might be interested in the differences between the app stores. Different stores require different amounts of identification from their users to buy and sell apps, or offer different terms and conditions.

The precise terms for using these app stores are hidden in user and developer agreements. The agreements hide within pages of legalese what the stores are *contractually* allowed to do with a user's data and a developer's app.

A comparison of the terms and conditions is given in Table 4.1 (starting on page 89). It is also worthwhile comparing the forms the terms and conditions themselves take. For most of the stores, the terms are single files or web pages written in English easily accessible from the store homepages [115, 9, 51, 6].

Apple's terms documents are split into many files and websites, and it is not immediately clear how each document is related. The user agreements for all app software and hardware are kept on one site in hierarchical menus with each product, version of a product, and for each country appearing to have a separate terms document. In practice many products share the same linked contracts, within a single country. Developer terms and agreements are kept in a different website and presented as a list of eleven policies. The iTunes terms and conditions, which form the bulk of the agreements for using the iOS app store. The iTunes terms and conditions are even available in comic-book format [93].

Comparing the terms and conditions in Table 4.1, the stores are for the most part rather similar. Stores pay their developers a month in lieu. A user must be over 18 to use the store, unless their guardian permits it. No store is okay with developers selling illegal or sexually explicit material. Most stores are going to record information about the users who bought software from them. One area where the stores do differ is in terms of refunds—only Apple has a return period longer than a few hours. In the case of Amazon there is no right to a refund, and for Yandex only 15 minutes.

Another area where they differ is the rights a store has to the apps they sell. Amazon and Aptoide both have the right to alter the apps. This is important because it changes the trust model for apps on Android. In Android the developer who signs the app is responsible for updates and identified as the developer of the app. Android will not let a user install unsigned apps[3]. To modify an app, Aptoide and Amazon must be able to re-sign the code, essentially taking on the role of vouching for the apps integrity and, by implication, its safety and security. Whereas an app from Google's marketplace would be signed by the developer who made it, the Amazon app store signs the apps it

[3]Normally. There are exceptions for development.

	Apple	Amazon	Aptoide	Google	Yandex
App User identification	Apple ID.	Amazon account.	An ID and contact details. User agrees not to give false ID.	Name address and billing details.	For free apps no ID is needed, For paid apps they must be an *authorised user* and give payment details
Taken information	Technical data about the device, system, software and peripherals; which the Application Provider may use provided it is in a form that does not personally identify the user.	Device info, network connectivity, download info, location, usage info. They will not access information unrelated to the app store.	Transaction information, which may be shared with developers.	App installation data (for malware identification), which may be opted out of. Device ID, visited URLs and cookies.	Device info, OS info, device content, apps and services. Mobile and SIM information. Location (which can be denied), search queries and technical information.
Payment info	Credit card or gift card.	Through Amazon.	Through an approved partner's payment processor.	Google Wallet, others at Google's discretion.	Through an approved processor provided by Yandex.
Who pays whom?	The purchasing user pays Apple. If a user uses *family sharing* then the family organiser pays Apple.	User pays Amazon.	User pays the store owner through Aptoide.	User pays Google Commerce or the provider where Google is acting as an agent.	User pays the app supplier.
Pricing	Apple sets price tiers which a developer may choose from.	Amazon, based on a minimum or suggested price from the developer.	The developer and store owners. Aptoide may round prices.	The developer. Google may round prices. If the developer makes an app free they may not charge later.	The supplier. The supplier agrees that Yandex may limit the exact amount of money and may convert to other currencies.
Refunds	14 days.	No.	24 hours.	Only for defective or removed content. A user may ask for a refund for 2 hours. For an amount less than 10 USD a refund may be issued up to 48 hours later. For amounts above 10 USD it depends on who processed the payment.	15 minutes.
Age of use	13 or older to create an Apple ID. Parents or guardians can create an ID through family sharing. Educational institutions can create IDs for students.	Bellow 18 with consent of parent or guardian. No alcohol content for under 21s.	A legal age within the user's country.	At least 13. Bellow 18 with parent or guardian's consent.	At least 14. Bellow 18 with parent or guardian's consent.
Updates	Apple are not obliged to provide updates, and user's can chose not to install them.	By default they will be installed.	Aptoide can check for updates to apps and users will receive them.	Google will check for updates to apps and users will receive them.	Yandex may update content on devices for security and bug-fixing reasons.

	Apple	Amazon	Aptoide	Google	Yandex
Moderation	Apple will moderate based on their own opinion of what is appropriate.	Publisher is obliged to give information which may be used to give ratings. Amazon cannot check these ratings are correct.	Aptoide may review, screen, change and review apps but are not obliged to. The Aptoide *trusted app* sign indicates an automatic anti-virus checker checked the app and does not imply Aptoide checked it.	No obligation but Google may.	They may moderate and filter but are not obliged to.
Rights to content	Apps are licensed to a user and not sold. The user may use content on Apple devices. If the device is later sold the user must remove the content. The user may not distribute, copy, reverse engineer, disassemble, change or create derivative works with the licensed application or part thereof.		The user may not change, rent, lease, loan, sell, distribute or create derivative works based on the content. The user may use the content.	The user cannot use apps as part of a public performance. The user may transfer apps between linked Google accounts. The user may not use content for dangerous activities, such as nuclear plants, life supporting, emergency communications, aircraft control or similar activities where failure might lead to death, injury and environmental damage.	
What can the store do with the app?	Apple are appointed as the developer's agent/commission-aire to sell, market, and deliver apps. Apple may *thin* an app, and repackage it for certain devices and optimisation.	Irrevocable royalty-free right to distribute through all electronic means. Amazon can check, test and store the app. Amazon may change and add to the app for analytics, policy enforcement, to add metadata and to improve compatibility with Amazon devices. They may, if requested by the developer add DRM. Amazon may use the app to promote and advertise their services, and may make limited time promotions or *test drive* versions of the app. They may claim other rights.	Sell and make available to third-party stores. They may change the app.	Google can reproduce, perform and display the app for marketing purposes. Google may delegate this right to a third-party.	Yandex may use the app and developer's marks, logos and images of the app and its content worldwide. Yandex may delegate or sub-license this right.

	Apple	Amazon	Aptoide	Google	Yandex
How quickly will they remove an app at a developer's request?	No time period specified, but the developer may ask that they do it.	10 days generally. 5 days if due to a loss or third-party claim to rights. ASAP if it is due to breaking the law.	No time period specified, but the developer may ask that they do it.	No time period specified, but the developer may ask that they do it. If it is due to IP or breaking the law then users who downloaded within the last year can have a refund.	90 days, though Yandex may keep an archive copy.
Additional EULAs	The developer may add one, but it may not be inconsistent with the minimal terms and conditions provided by Apple, and must comply with all applicable laws in the regions the app is sold.	The developer may add one if it doesn't interfere with Amazon's own terms.	Aptoide add a default one, but they suggest developers add one that supersedes it.	Developer may add one, else the user is granted the worldwide perpetual right to perform, display and use the product.	Developer must add one and it must include the right to the content worldwide (except for trials).
Developer identification	Apple Developer ID.	Amazon account.	Email address. Using the same email as used for a Google Play developer account may help speed the app review process.	Google account and payment info.	Email, company name, tax ID, addresses, country of residence, website, order email address, user support email address, urgent Yandex support email address, payment information, other *reasonable information*, which they do not define further.
Content restrictions	Described by a *living document* [7], which Apple will change as new example come up. Broadly they look for safety, performance, business, design and legal issues. They also are aware *kids* use their store and want developers to think about them when submitting apps. They will reject amateurish, cobbled together apps. Apps that they believe to have content that is *over the line* (they say *"they will know it when they see it"*). Attempts to cheat the system will lead to developers being banned.	No offensive content, pornography, illegality, gambling with real currency, IP infringing, privacy infringing, copyright infringing, or content that would be illegal in which the country the app would be sold.	Nothing that displays or links to: illegal content, invasions of privacy, content that interferes with the services of others, hate or violence, IP infringement. Nothing that harms devices or personal data. Nothing that has unpredictable network usage or has an adverse impact on a user's service charges or a carrier's network. Nothing that creates a *spammy user experience*.	No apps implementing alternate stores can be submitted to the store. No sex or violence, bullying, hate, impersonation, IP infringing, PII publishing (specifically credit or ID card information, or non-public contacts), illegal content, gambling, dangerous (malware or spyware), self-modifying or system interfering content. No unpredictable network use.	Content must be safe, free of defects, and described accurately. Nothing disruptive to Yandex or malware. Nothing illegal such as: child pornography, obscenity, nudity, sex, extremist, hatred, violent, discriminatory, defamatory, gambling, copyright infringing or other forbidden material. Nothing that steals private information nothing that mimics system functionality. Nothing that would require Yandex to open-source anything via copy-left. No alternative marketplaces.

	Apple	Amazon	Aptoide	Google	Yandex
Who describes the apps	Developer	Developer but Amazon may edit it.	Developer.	Developer.	Developer but Yandex may edit it.
Who can distribute apps?	Apple.	Amazon, and regional subsidiaries.	Aptoide and third-party partners using Aptoide to create their own store.	Google.	Yandex and partners.
What support must a developer give?	None, except what is in app's EULA. The developer must acknowledge that they are solely responsible for maintaining their app (not Apple).	Must respond to users within 5 days. Must respond to Amazon within 24 hours if Amazon deem issue *critical*.	Aptoide will give users with the developer's contact information.	Developer must support their app and handle complaints. Developer must respond within 3 days and to issues deemed urgent by Google within 24 hours. Failure to do so will result in Google lowering your app's ranking, review scores, and removal.	Must give user with support via email or phone. Must respond to users within 5 days.
When do developers get paid?	Within 45 days of the last day of the month. Minimum earned balance is either 10 USD or 150 USD, depending on the developer's bank's country.	Roughly 30 days after app was sold.	Roughly 30 days after the end-of-the-month of the purchase.	On the 15th of the following month via Google Wallet. Minimum earned balance for local currency payout is 1 USD. 100 USD for wire transfer payouts.	30 days after the end-of-the-month of the purchase. Minimum earned balance for payout is 100 USD. If less is earned in a month Yandex will store it without interest.
What do developers get paid?	Roughly 70% of the price, as specified by [8].	70% of list price (minus card processing fees).	75% of revenue share after deduction of all transaction fees. Other rates subject to agreement with Aptoide.	70% of payment by user.	70% of net revenue (minus transaction fees).

Table 4.1: Comparison of terms and conditions from five app stores.

sells. In theory, an app originally purchased in the Google Play store could be updated by an app from the Yandex store (if signed by the same key and the app was a later version). With Amazon's model this is not possible as they sign the app with different key. In Amazon's case this is especially confusing as Amazon manufacture their own Android tablets. To access certain *system* permissions Android requires that an app be signed by the same key that signed the OS installation. In the Play store model Google no third-party developer in their store would have access to that key, but in the Amazon model they could (if they desired) distribute system apps for their Kindle devices. Having a different signing model is not inherently wrong, but it changes the trust assumptions associated with Android, which a user may not necessarily be aware of. We

```
'apple' says 'maximum-purchase-hours' is(336).
'apple' says User:U canRequestRefund(Purchase:P)
  if U hasCompletedPurchase(P), 'maximum-purchase-age' is(N)
  where age(P) < N.

'aptoide' says 'maximum-purchase-hours' is(24).
'aptoide' says User:U canRequestRefund(Purchase:P)
  if U hasCompletedPurchase(P), 'maximum-purchase-age' is(N)
  where age(P) < N.

'google' says 'maximum-purchase-hours' is(2).
'google' says User:U canRequestRefund(Purchase:P)
  if P isFor(App:A), A isDefective,
     U hasCompletedpurchase(P), 'maximum-purchase-age' is(N)
  where age(P) < N.

'yandex' says 'maximum-purchase-hours' is(0.25).
'yandex' says User:U canRequestRefund(Purchase:P)
  if U hasCompletedPurchase(P), 'maximum-purchase-age' is(N)
  where age(P) < N.
```

Figure 4.3: AppPAL translations of app store refund rules.

discuss this further in Section 4.1.2.

The terms and conditions are dense technical documents. Yet despite the document's intimidating nature, the differences in terms between each app store agreement is often small. By encoding the key differences into a formal language (such as AppPAL) we can clarify the distinctions between the stores. A developer, or user, can then choose the app stores they are happy to submit to based on the store's policies, and a company could automate the enforcement of the policy. For example, in the case of refunds the differences in policy could be written as shown in Figure 4.3. The differences in support agreements is shown in Figure 4.4. Again the differences are small: only differing in the number of hours developers have to respond and whether there is a need to respond to issues from the store itself more urgently.

Writing policies out in this way describes the informal terms and conditions unambiguously. A computer could check compliance with the terms by evaluating the policy in this form (assuming it was given sufficient information).

```
'amazon' says Developer:D mustRespondWithinHours(Issue:I, 120)
  if User hasIssueWith(I, App), D hasCreated(App).

'amazon' says Developer:D mustRespondWithinHours(Issue:I, 24)
  if 'amazon' hasIssueWith(I, App), D hasCreated(App),
    I isCritical.

'google' says Developer:D mustRespondWithinHours(Issue:I, 72)
  if User hasIssueWith(I, App), D hasCreated(App).

'google' says Developer:D mustRespondWithinHours(Issue:I, 24)
  if 'amazon' hasIssueWith(I, App), D hasCreated(App),
    I isCritical.

'yandex' says Developer:D mustRespondWithinHours(Issue:I, 120)
  if User hasIssueWith(I, App), D hasCreated(App).
```

Figure 4.4: AppPAL translations of app store support rules.

If we were to translate the entire app store terms and agreements we could build a set of standard decisions that describe what app stores terms and conditions talk about. The similarity between the different rules in Figure 4.3 and Figure 4.4 suggests there are idiomatic patterns present in the policies that could be captured and described. In Chapter 5 we look at BYOD policies and identify decisions and idioms common to those policies on the basis of an AppPAL description of the rules. Similar techniques could also be applied to app store terms to clarify and describe the policies precisely; going beyond the comparison of the text of the policies presented here.

4.1.2 Why Who Signed the App Matters

Some stores may make changes to the apps they sell. For Android users this is important as it changes the trust relationships around who can provide updates and who developed the app. We will illustrate the decisions Android makes using AppPAL.

To install an app Android requires it to be signed with a key to identify the developer who built the app:

```
'android' says App:A isAcceptable
  if A isSignedWith(Key),
     Key isValid.
```

The standard Android trust model says that the developer of the app should be the person that signed it:

```
'user' says Developer:D hasMade(App:A)
   if A isSignedWith(Key),
      Developer isOwnerOf(Key).
```

When upgrading an app the upgrade is only accepted if the certificates match (and the version number is higher). Additionally, if two apps have been signed by the same key, they will share the same process space (enabling them to share memory) and be able to share code:

```
'android' says App:A canUpgrade(App:B)
  if A isSignedWith(Key),
     B isSignedWith(Key),
     A hasVersion(V1),
     B hasVersion(V2),
     Key isValid
  where V1 > V2.

'android' says App:A canAccess(Data:D)
  if D isOwnedBy(DataOwnerApp),
     A isSignedWith(Key),
     DataOwnerApp isSignedWith(Key).
     Key isValid
```

To submit an app to the Play Store Google requires that the key the app was signed with must be valid until 2033, and more generally they recommend keys are valid for at least 25 years[4]:

```
'play-store' says App:A isAcceptable
  if A isSignedWith(Key),
     Key isValid
  where expiryDate(Key) > 2033.
```

If an app is re-signed by a store to alter an app, then it loses the relationships set up by its original key. The store must now provide (or re-sign) updates

[4]The validity of the key is somewhat surprising. We believe that this is to ensure the key outlives the app and so avoid issues with the developer's keys expiring whilst they are still providing updates to the app.

for the app it sold. If the new key is also used to sign other apps then the app may leak (or take) information from other apps shared by the same key. Assumptions that data is private to one particular app start to break down if apps are re-signed. The user's trust model must now become:

```
'user' says Store:D hasSold(App:A)
  if A isSignedWith(Key),
    Developer isOwnerOf(Key).
```

This policy is equivalent to the user's original trust model (they rely on the same decisions), but the meaning is subtly different. The key that signed the app is no longer sufficient to identify the developer.

4.2 Finding the Right Apps

For a user, finding the right apps is tricky. Users need to discover which ones are not going to abuse their data. This is difficult as it isn't obvious how apps use the data each has access to (including data generated by the app or data already on the device). Consider a user attempting to buy a torch app. The Play store shows users a long list of apps when they browse. Clicking through each app they can find the permissions each requests but not the reasons why each was needed. They can see review scores from users but not from tools to check apps for problems and issues like SSL misconfigurations [47]. 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. In Android Marshmallow apps display permissions requests when the app first tries to access sensitive data (such as contacts or location information). In general, research shows that users do not understand how permissions relate to their device [48, 105]. Nevertheless, the device user must make the decision which apps to use and which permissions to grant.

Some apps are highly undesirable. One class of apps are called potentially unwanted programs (PUPs). They represent apps which whilst they are not exactly malicious, are designed to do things which may annoy users, such as displaying pop-up notifications for adverts or altering wallpapers on the device. There are many PUPs being sold for Android devices [107, 104]. Employees are increasingly using their own phones for work and an employer may restrict

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

Table 4.2: Lin et al. privacy preference policies expressed as sets of prohibited permissions.

which apps their employees can use. The IT department may set a *mobile device policy*—a series of rules describing what kinds of apps employees can used and how—to prevent information leaks. Some users worry that 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 get access to their location, or address book; so they may apply their own personal, informal, security policies when downloading and running apps.

4.2.1 Privacy Preferences

To what extent do users follow personal policies informally? In a study of 725 Android users, Lin et al. found four patterns that characterise user privacy preferences for apps [82], demonstrating a refinement of Westin's privacy segmentation [64].

Westin grouped the people into three categories based on their responses to surveys. *Privacy Fundamentalists* were highly concerned with their privacy and actively distrusted businesses governments and technology with their personal data and disputed any need from these groups to collect the data. *Privacy Unconcerned* people, in contrast, were not concerned with giving a way information and who trusted others to not mismanage information about them. The third group, the *Privacy Pragmatists*, were more considered and would have some personal policy that described when they were happy to give up (or protect) their privacy.

Lin et al. identified four types of user when assessing user's personal app privacy policies. The *Conservative* (C) users were uncomfortable allowing an

Westin Privacy Segment	Lin et al. Privacy Preference
Privacy Fundamentalist (25%)	Conservative (12%)
Privacy Pragmatist (55%)	Advanced (18%) Fencesitter (22%)
Privacy Unconcerned (20%)	Unconcerned (48%)

Table 4.3: Lin and Westin privacy groups and their sizes.

app access to any personal data for any reason. The *Unconcerned* (U) users felt okay allowing access to most data for almost any reason (though they were somewhat uncomfortable about allowing apps unrestricted access to their accounts). *Advanced* (A) users were comfortable allowing apps access to location data but not if it was for advertising. Opinions in the largest cluster, *Fencesitters* (F), varied but were broadly against collection of personal data for advertising.

Lin et al.'s four groups roughly map to Westin's three, with *Conservatives* being *Privacy Fundamentalists*, *Unconcerned* users being *Privacy Unconcerned*, and the *Advanced* and *Fencesitter* users together acting as the *Privacy Pragmatists.* A criticism of Westin's privacy groups is that he did not distinguish between pragmatic people and those who gave vague responses in the *Privacy Pragmatists* group [110]. Lin et al.'s groups seem to support their *Advanced* and *Fencesitter* groups seem to cover both aspects of the *Privacy Pragmatist* group. Lin et al. and Westin's groups differ, however, in terms of size. In general, Lin et al. found more unconcerned users, and less fundamentalist or conservative users then Westin estimated existed in the general population (Table 4.3).

Question	To what extent do user's follow an app policy?
Input	Policy to test. Database of users and which apps they had installed. The installed apps.
Output	The percentage of each user's apps they had that satisfied the policy.
Method	1. Express policy using AppPAL in the form of: `'researcher' says App:A hasMetPolicy('policy-name') if ...` 2. For every installed app, check whether: `'researcher' says 'app-name' hasMetPolicy('policy-name').` 3. For each user count then number of their apps for which the policy was satisfied. The output is the percentage for which the check came back as true.
Hypothesis	If a user is following a policy, we would expect most of the apps they installed to meet the policy.

Table 4.4: Summary of experiment to measure the extent user's follow a policy.

4.2.2 Measuring Users

To answer the question of to what extent do users follow personal privacy policies in practice we performed the following experiment (summarised in Table 4.4). We wrote AppPAL policies to describe each of Lin et al.'s behaviours as sets of prohibited permissions, shown in Table 4.2. For example the *Fencesitter* policy is encoded as:

```
'researcher' says App:X hasMet('fencesitter-policy')
  if X isWithoutPermission('GET ACCOUNTS'),
     X isWithoutPermission('ACCESS FINE LOCATION'),
     X isWithoutPermission('READ CONTACT').

'researcher' says App:X isWithoutPermission(Permission:P)
  where check_permission(X, P) = false.
```

Where `check_permission` is a constraint that checks whether the app `X` requests permission `P`. Each policy in Table 4.2 was translated into a similar AppPAL rule where the permissions that made the users uncomfortable were prohibited. These simplify the privacy policies identified by Lin et al. as we do not take into account the reason each app might have requested each permission (we could write more precise rules if we knew why each permission was requested).

Lin et al. used Androguard [40] as well as manual analysis to find the precise reasons the app requested each permission [82].

We took installation from a partially anonymised database of installed apps captured by Carat [90]. This allowed us to link users to the apps they had installed and measure the extent each user was following the policy.

The Carat data set was collected as part of a UC Berkley and University of Helsinki experiment to measure the energy usage of apps. As part of the experiment, users of the Carat app allowed the app to collect which apps were installed on their device. The data set was anonymised by replacing the app names with hashes of the apps package identifier, and usernames with an increasing integer key.

The app names were replaced with hashes in order to obscure the package names of some apps. We spoke to one of the researchers who collected the data and learnt that the Carat tool was tested inside a company who were developing some apps that had not been announced. The company did not want to leak the names of their private apps so they were hashed in order to preserve their secrecy. This allowed researchers to reverse engineer the hashes of publicly known applications, whilst keeping any secret or unknown applications private.

Having confirmed with the data-set owners that reverse engineering the app hashes would not raise ethical concerns, we used John the Ripper [103], with a database of known package names and hashes, mostly derived from the Android observatory [19] to link some of the app hashes to app names. Using the reverse engineered hashes we can see which users (identified by a number) had installed which apps.

The database has over 90,000 apps and 55,000 users. On average, each Carat user installed around 90 apps each, and 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 variation 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 [19], we checked which apps satisfied which policies.

To check which apps satisfied which of Lin's policies, we took our AppPAL encoding of the policies, and facts identifying the apps in our data-set actual apps into an assertion context.

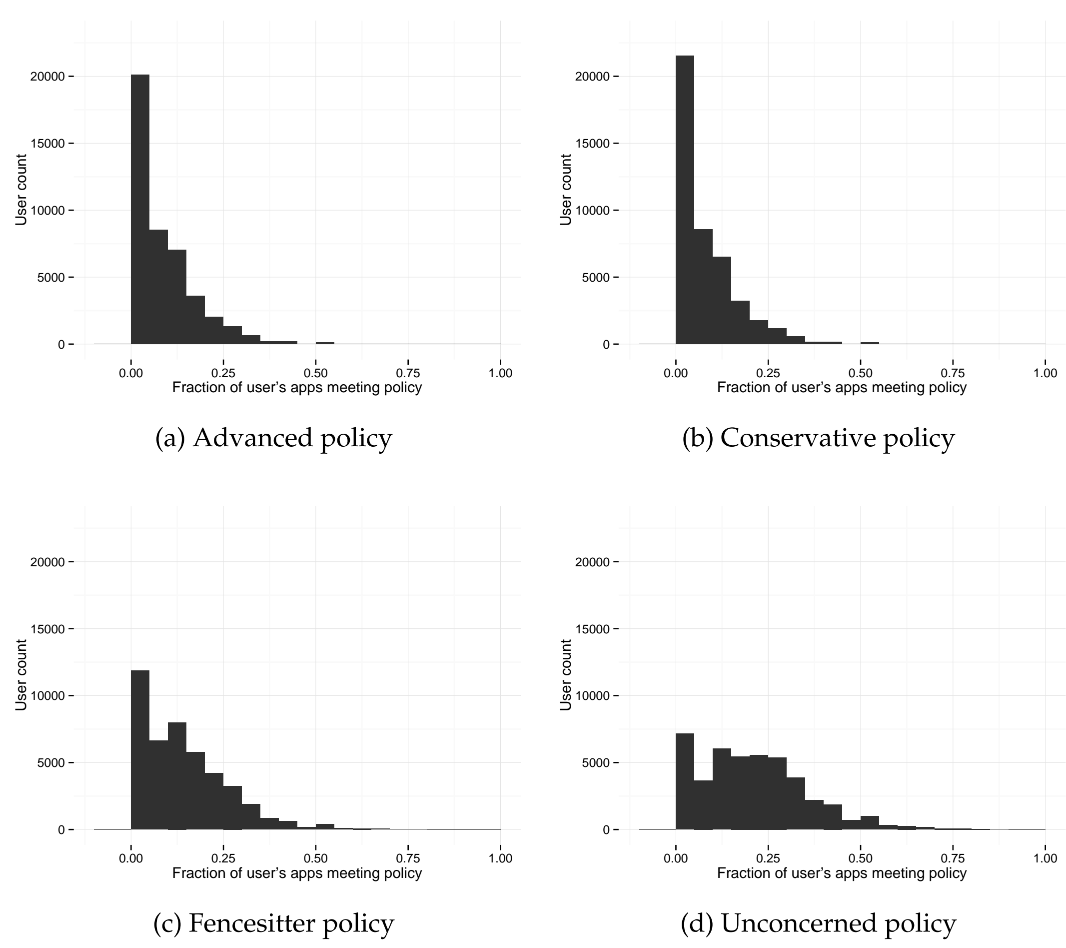


Figure 4.5: Adoption of the four Lin et al. policies among users from the Carat data set. Most users do not appear to follow these policies most of the time.

```
'researcher' says App:X hasMet('unconcerned-policy')
  if X isWithoutPermission('GET_ACCOUNTS')

'researcher' says 'com.facebook.katana' isApp.
```

For each app, we then queried AppPAL whether the app had met each of the policies, and recorded which apps met which policies. This allowed us to measure the extent any of the users appeared to be following any of the policies, as the percentage of the apps they had installed that met the policy. Our results are shown in Figure 4.5.

We hypothesised that if a user's were following a policy, then we would expect the majority of the apps they had installed to meet the policy. In the case of the Lin et al.policies we found that very few users had a majority of apps which met the policy. This would suggest that user's may not be following Lin et al.'s policies in practice. The charts can be read as described in Figure 4.6. Each bar of the histogram represents a statement about what percentage-range of a users apps met a given policy. A user for whom between 0–5% of their apps met the policy would count towards the first bar, a user for whom between 5-10% of their apps met the policy would count towards the second bar. The height of each bar represents the number of users in each group.

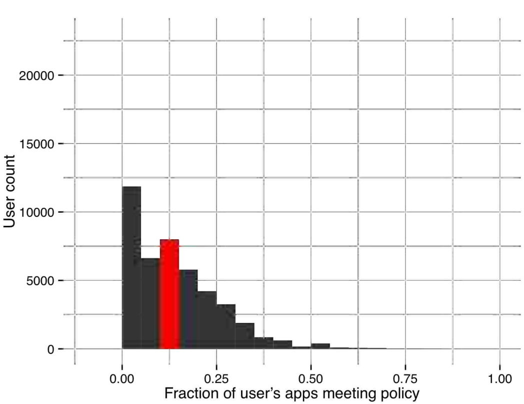


Figure 4.6: Chart highlighting that there were around 7,500 users for whom only 10-15% of their apps satisfied the fence-sitter policy.

Even for the least onerous *unconcerned* policy, most users did not seem to follow the policy most of the time. This suggests that user's privacy preferences are somewhat disconnected from their behaviour (assuming the user population studied by Lin et al. was similar to the data in the Carat study). This is reminiscent of the *privacy paradox* that states that whilst people will say they are very concerned about their privacy, they do not alter their behaviour to protect it. It was first noticed by psychologists looking at how people use social networks; though has appeared in many other areas since. A few users, however, did seem to install apps meeting one of these policies most of the time. This suggests that while users may have privacy preferences most are not attempting to enforce them. This suggests that policy enforcement tools, like AppPAL, may help users enforce their personally desirable policies which they cannot do easily using the current ad-hoc manual means available.

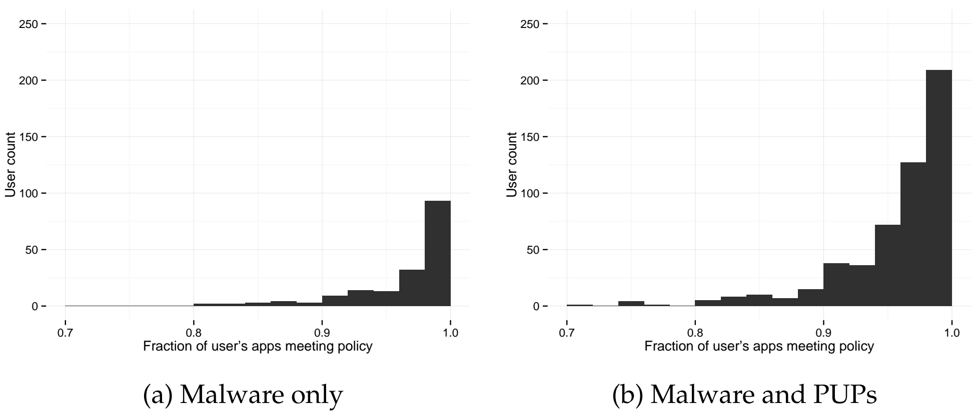
4.2.3 Privacy Policies and Malware

It is also interesting to discover whether people install apps classified as malware. McAfee classify malware into several categories, and provided us with a data set of apps classified as malware and PUPs. Using these package IDs we calculated the package hashes and looked to find users in the Carat data set who had installed any of these apps. The McAfee data classified the data into several categories based on the purpose of the malware. The *malicious* and *trojan* categories describe traditional malware. Other categories classify PUPs such as aggressive adware. Using AppPAL we can write policies to differentiate 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 be more selective about the apps they install. If a user is being selective we might expect them to install less malware and PUPs, as they often request many permissions. We can check this by using AppPAL policies to measure the amount of malware each user had installed.

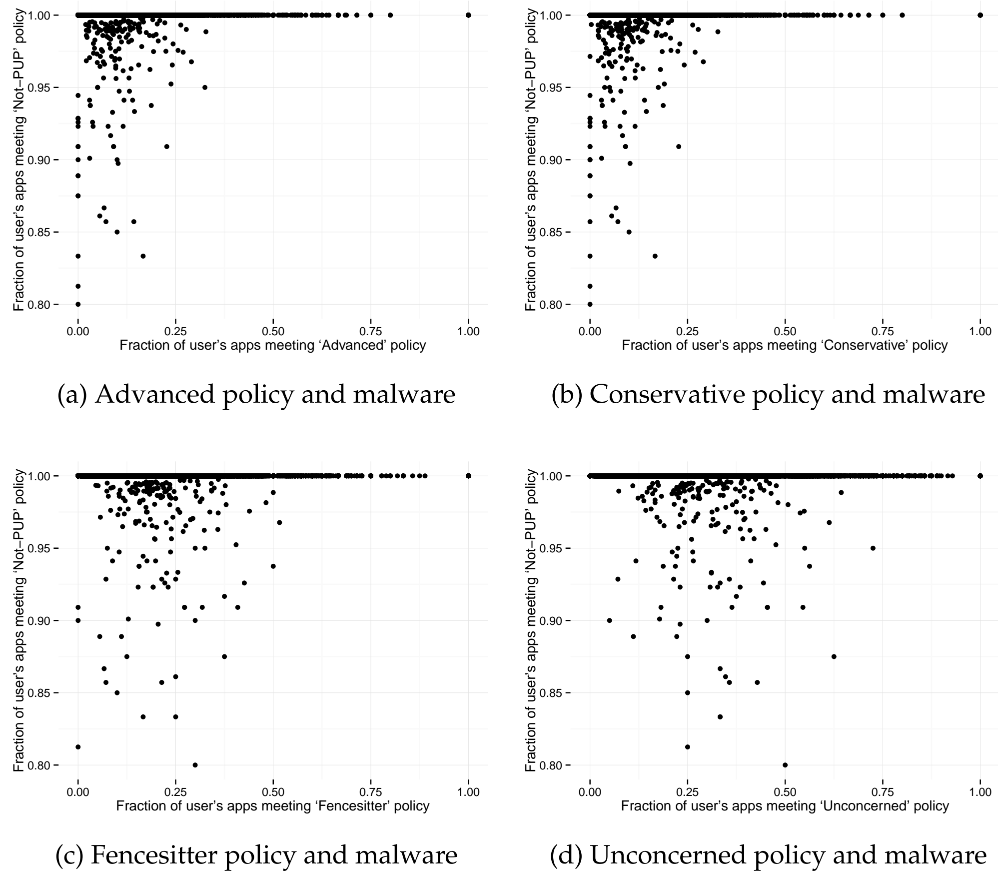
We found that 1% of the users had a PUP or malicious app installed. Figure 4.7 shows that infection rates for PUPs and malware is low; though a user is 3 times more likely to have a PUP installed than malware. It is interesting to compare a user's compliance to the Lin et al. policies with the amount of malware each had installed (Figure Figure 4.8). Users who were complying more than half the time with the conservative or advanced policies complied with the malware or PUP policies fully. This suggests that policy enforcement is worthwhile: users who do enforce policies about their apps experience less malware. This could also be attributed to the users selecting their apps more carefully to enforce their policy: a careful user is unlikely to install malware generally.



(a) Malware only

(b) Malware and PUPs

Figure 4.7: Malware installation numbers in the Carat data set.



(a) Advanced policy and malware

(b) Conservative policy and malware

(c) Fencesitter policy and malware

(d) Unconcerned policy and malware

Figure 4.8: Graphs plotting a user's conformance with the Lin et al. policies against the amount of malware they had installed on their device. Each dot represents a user.

4.2.4 Discussion

In capturing the privacy preferences, and in comparing them to user installation data, we have shown that most users use apps irrespective of how uncomfortable they are with the permissions the app request. A small set of users do seem to enforce these policies at least sometimes, however. Our comparison is not without some limitations:

- We do not have the full user purchase history, and we can only find out about apps whose names match those in available databases. So a user may have apps installed that break the policy without us knowing.
- The 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 [111], so a user may actually be more conservative than our analysis suggests.
- The Android permission system has changed over time. In particular, our app installation data comes from before users had the ability to revoke individual permissions from apps. Android's current permissions model makes it easier for a user to follow a policy as they do not have to chose between going against their privacy preferences and not installing an app. If we were to repeat the study with modern data, and looking at which apps had their permissions requests denied, we might expect to see more users following a policy (if users do wish to follow a privacy preference most of the time).
- The Carat data set is not the same group of users, as was examined by Lin et al.and so is not the same sample of users. In particular, the users in the Carat data-set had all installed the Carat app to monitor their power usage. We found that some users in the Carat data-set had also installed tools such as Busybox[5]. This suggests the Carat data-set contains more technically minded users than we might expect to find in the general population.

To avoid most of these limitations we would need to collect our own data, and work with users to better understand and capture their privacy preferences. We leave this to future work, however.

[5]Which gives the user access to common UNIX utilities such as a shell.

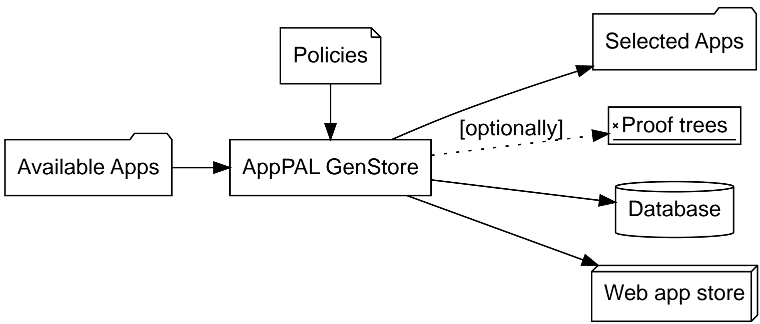


Figure 4.9: AppPAL GenStore's architecture. In go policies and apps, out come app stores.

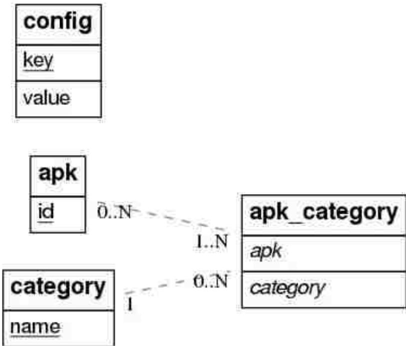


Figure 4.10: GenStore database schema.

4.3 An AppPAL Enhanced Store

Using AppPAL we can describe users preferences for apps, and we can describe the differences between some stores. A natural continuation of this is to start to generate new app stores, based on a user's policy, that only sell apps that a user might find acceptable.

Curated app stores are a similar idea, used inside companies to help employees install apps for work that satisfy company policies. With these corporate stores the apps are hand selected by IT staff.

The *GenStore* tool uses AppPAL to automate the store creation process for Android app stores. The architecture is shown in Figure 4.9. We summarise it as:

1. The developer provides a pool of apps to GenStore. GenStore will select from among these apps to create the curated store.

2. The developer also provides a series of policy files. These should include rules to decide if an *App isSellable*. Optionally they can also describe predicates where *App hasCategory(Category)* which will be used to organise the apps into categories in the store.

3. GenStore runs, and for every app checks whether it is *sellable*. Additionally, it checks apps for each category (apps can belong to multiple categories), if defined. From this the GenStore builds a database (Figure 4.10) documenting which apps are available, their categories and some additional metadata.

4. A web-app is created from a template. The default template uses the Sinatra framework to serve the store, but any framework could be used. Alongside the web-app, Genstore copies the selected APK files into a directory and a database is created storing information about the apps.

4.3.1 Using GenStore to Build an App Store

As an example of GenStore consider Alice who wants to build a curated app store for her company. The apps in her store will be split into two categories: *required* apps which is a list of apps specified by the HR department that everyone should install, *optional* apps which people can install if they want, and *Alice's Favourites* which are apps Alice particularly likes and wants to show off. Apps which aren't in these categories shouldn't be available and, as well as these three categories, only apps which have been checked by the company antivirus program should be allowed on her store.

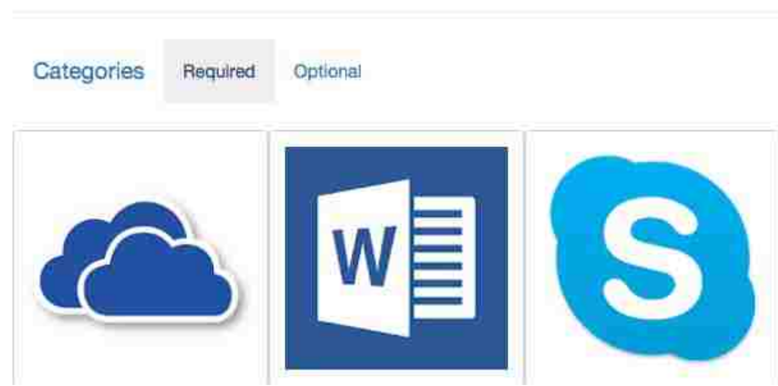
To implement her store she downloads a pool of apps that she could sell in her store, and writes a policy. In this case we chose a pool of 12 apps to serve as an example. The policy describes what she wants to sell in her store and how she wants to categorise the apps. This policy is very simple (it essentially just white-lists apps) but more complex policies could be written if Alice wanted.

```
'store' says 'alice' can-say inf App isSellable.
'store' says 'alice' can-say inf App hasCategory(C).

'alice' says App isSellable
  if App hasCategory(C),
  where AVCheck(App) = true.

'alice' says 'hr' can-say X hasCategory('Required').
'hr' says 'apk://com.microsoft.office.word' hasCategory('Required').
'hr' says 'apk://com.microsoft.skydrive' hasCategory('Required').
'hr' says 'apk://com.skype.raider' hasCategory('Required').

'alice' says 'apk://com.niksoftware.snapseed' hasCategory('Optional').
'alice' says 'apk://net.skyscanner.android.main' hasCategory('Optional').
'alice' says 'apk://com.google.android.apps.photos' hasCategory('Optional').
'alice' says 'apk://com.sega.sonicdash' hasCategory('Optional').
```

With the policy written, she passes the apps and the policy to GenStore and asks it to build her an app store. It finds that several of the 5 of the apps were not sellable in her store (4 because they did not have categories, and the Towelroot rooting tool was rejected by the antivirus), but that it could create an app store from the remainder. The output of the tool, and the web app it creates is shown in Figure 4.11. Alice can take this web app and modify it to work in her company.

4.3.2 Current Status

The GenStore tool itself is usable as a prototype, but it is not production-ready. The stores it creates are minimal and do not have any authentication; and as such further work would be required to make the stores ready for use in an actual company. Genstore lets developers explore how policies could be integrated into the Android ecosystem. It shows an example of how AppPAL can help make decisions, in this case about which apps should be included in a company app store, automatically. For a company seeking to enforce a policy about which apps are installable they could use it to offer their employees choice about apps, without having to remove the default store entirely. If an AV vendor integrated their own checks into AppPAL then they could create a *safe* app store where all apps have been demonstrably vetted by their software and engineers.




```
$ java -jar genstore.jar -d data -p store.policy
[INFO] preparing store
[INFO] picking apps
[INFO] com.microsoft.skydrive is sellable.
[INFO] com.niksoftware.snapseed is sellable.
[INFO] com.google.android.apps.photos is sellable.
[WARNING] com.geohot.towelroot is not sellable.
[INFO] com.sega.sonicdash is sellable.
[WARNING] com.rovio.baba is not sellable.
[INFO] net.skyscanner.android.main is sellable.
[INFO] com.microsoft.office.word is sellable.
[WARNING] com.supercell.clashofclans is not sellable.
[WARNING] com.whatsapp is not sellable.
[INFO] com.skype.raider is sellable.
[WARNING] com.outfit7.mytalkingtomfree is not sellable.
[INFO] picking categories
[INFO] com.microsoft.skydrive has category Required.
[INFO] com.microsoft.office.word has category Required.
[INFO] com.skype.raider has category Required.
[INFO] found 3 apps with category 'Required'
[INFO] com.niksoftware.snapseed has category Optional.
[INFO] com.google.android.apps.photos has category Optional.
[INFO] com.sega.sonicdash has category Optional.
[INFO] net.skyscanner.android.main has category Optional.
[INFO] found 4 apps with category 'Optional'
[INFO] building store
[INFO] copying app structure
[INFO] creating tables
[INFO] adding apks
[INFO] adding categories
[INFO] adding metadata
[INFO] copying apks
```

Figure 4.11: Output of the GenStore tool which runs on the command-line and produces a web app.

Chapter 5

Applying AppPAL to BYOD Policies

The relationships between users, apps and app stores is one part of the mobile ecosystem, but another aspect is what happens when user's start to bring their personal devices into the workplace. In this chapter we move from looking at personal policies to corporate ones. BYOD policies are one way a company might try to control how personal devices are used in the workplace. In this chapter we show how we can capture these policies using AppPAL. We focus on the trust relationships within these policies and identify two idioms, delegation and acknowledgements, that existing tools for enforcing BYOD policies ignore.

5.1 BYOD in the Workplace

Many employees bring their personal mobile devices to work. To control the access these devices have to company resources an estimated 72% of companies publish BYOD policies. According to a survey of companies by LinkedIn [101], 40% made BYOD available to all employees and 32% made it available to selected employees. These BYOD policies are natural language documents that employees should read and obey. They describe steps to take to secure devices in the workplace. The policies say how employees should get access to data, and who should authorise decisions.

Companies can use their policies in varying ways. Some may trust employees to follow the rules on their own. Others may use MDM software to implement part of their policies. Tools such as IBM's MaaS360 and Blackberry's BES [67, 25]

can configure devices to restrict functionality and manage apps.

Commercial tools have limits to what they currently enforce. Some tools can only enable simple on-off configuration settings, and ban explicitly black-listed apps. More advanced systems can use app-rewriting to recompile apps. This lets them tunnel traffic through a VPN, or use geofencing to apply policies in predefined areas. These tools are not infallible. One survey found that 50% of companies with MDM software still had non-compliant devices in their networks [84]. Whilst app wrapping can protect some apps, in general it is ineffective [63].

5.1.1 Overview of Five BYOD Policies

This chapter looks at how we can apply AppPAL to BYOD policies. To do this we analyse of five *real-world* policies. We chose the policies from a variety of domains. They cover a range of different policy styles and concerns. Two are example policy templates a company might want to base its own BYOD policy on, published by security advice organisations. The remaining three are BYOD policies used inside companies.

The *Security Policy Template* [52]. It was published by the SANS Institute. This policy is a hypothetical policy published to help companies mitigate the threats to corporate assets caused by mobile devices. Companies change the document to suit their needs and BYOD requirements. The policy is general; not specific to any particular industry, device, or country's legislation.

The Health information and Management Systems Society [65]. HiMSS is a US non-profit company trying to improve health care through IT. The Health information and Management Systems Society (HiMSS) policy is short and has concerns specific to healthcare scenarios. The policy is a contract the users follow. It has a different style to other policies: in the other policies *the company states* what users should do; here *the user states* what they will do. The policy is a sample agreement for a healthcare company managing mobile devices.

A British hospital trust [73]. It describes the BYOD scheme used in practice at the hospital. The policy is broad and covers rules for corporately owned devices. This policy also briefly describes how devices should interact with patients.

The University of Edinburgh [112]. It is brief. It groups rules into those for high and low risk users. The policy is extremely general describing rules for laptops, tablets and phones. Many of the rules are vague. The policy says *"Use anti-virus software and keep it up to date"*, leaving the choice of AV software and update times to the user.

A company selling emergency sirens [35]. Again this policy is simpler, and, like the Edinburgh policy, is rather general compared to the other policies.

Each policy is split into a series of rules employees should follow. Typically, the policy is describing what employees should do and what will happen under certain circumstances. For example the NHS policy states that:

> **NHS**: *In the event of loss of the device, all data including apps will be wiped. The Trust is not responsible for reimbursement of any costs for personally purchased apps.*

The Siren policy matches this style. It states what conditions will lead to the company wiping an employee's device:

> **Sirens**: *The employee's device may be remotely wiped if: • The device is lost or stolen. • The employee terminates his or her employment. • IT detects a data or policy breach, a virus or similar threat to the security of the company's data and technology infrastructure.*

The HiMSS policy has a different style. The equivalent rule is phrased from the perspective of the policy subject (*"I agree. . . "*):

> **HiMSS**: *I agree that the PDA/Smartphone can be wiped by XYZ Health System upon the decision of XYZ Health System management and understand that it will delete all data including personal files.*

The SANS policy is written in the same style as the NHS policy. It is a template policy that a company might use as the basis for their own policy. Sometimes, however, it says things that seem like advice to the IT department for what to look for. For example:

> **SANS**: *A corporate mobile device management solution SHALL feature remote device wiping (or possibly only blocking) mechanism for all devices accessing corporate internal networks.*

The SANS policy also distinguishes between rules that should always be followed (SHALL), and those that may depend on a specific business's situation (SHOULD):

> **SANS**: *Basically, sentences using the verb "SHALL" are mandatory requirements applying to practices with high probability of putting the business at risk, whereas "SHOULD" means that the policy needs to be applied according to the business's specific situation.*

The Edinburgh policy goes further, grouping rules by device type and giving a security level to each rule. The policy expects high and medium risk users to follow everything, and low risk users to consider following the rules marked with a ∇. For example, the policy groups together the rule for remote wiping devices with the rules specific to mobile devices and tablets:

> **Edinburgh**: ∇ *Configure your device to enable you to remote-wipe it should it become lost.*

Our translation shows the text of the policy next to the AppPAL version. We have included the full translations in Appendix A. The rest of this chapter looks at what we found from the translation process, and comments on MDM policies in general.

5.1.2 Review of MDM software

A company might use MDM software to enforce their rules. Many vendors offer MDM solutions: including IBM (with their tool *MaaS360*), VMware (*AirWatch*) and MobileIron (whose tool has the same name as the company). They all offer a similar set of features, namely:

1. App wrapping. The tool modifies apps to offer some extra features or network properties. This may be limited to routing all the app's network traffic through a VPN.

Feature	MaaS360	Blackberry BES	MobileIron	Citrix XenMobile	VMWare AirWatch	Microsoft	SOTI MobiControl	Sophos	Landdesk
Antivirus								✓	
App selection/store/management	✓	✓	✓	✓	✓	✓	✓	✓	✓
App wrapping/modification		✓	✓	✓	✓	✓		✓	✓
Authentication	✓	✓	✓	✓	✓	✓	✓		
Compliance reporting	✓	✓	✓	✓	✓	✓	✓	✓	
Device configuration	✓	✓	✓	✓	✓	✓	✓		✓
Email/Calendar/Contacts/Documents	✓	✓	✓	✓	✓	✓	✓	✓	✓
Feature Restrictions	✓	✓			✓	✓			
Licence distribution		✓							
Location based settings	✓				✓				
Network configuration	✓	✓	✓	✓	✓	✓	✓		
Password/Encryption settings	✓	✓	✓	✓	✓	✓	✓	✓	
Remote wipe	✓	✓	✓	✓	✓	✓	✓	✓	✓
Security auditing	✓	✓	✓	✓	✓	✓	✓	✓	
Tracking/Spyware	✓	✓				✓	✓		✓
Watermarking					✓				

Table 5.1: Summary of different MDM software's capabilities built from information from each of the tools sales pages.

2. Basic security configuration. Lets an administrator to turn on and off various settings and security features. These include WiFi settings, passcodes, encryption or Bluetooth. These features could be used to reduce the attack surface of a device; or enforce policies that need a user's location by turning on the GPS.

3. Provisioning. IT departments can install and update apps and their configuration files. Email and LDAP configuration is common.

4. A curated app store. IT departments can white-or-black-list apps, and offer them to employees.

Table 5.1 summarises the features of 9 competing MDM packages identified by a Gartner report [102]. Most of the tools link to the report on their homepages as the report lists *every* tool as either *visionary*, *leading*, or *niche player*. The meaning of each term is unclear, however.[1] The tools are similar with the main difference being the UI and some extra features only some tools have.

[1] Table 5.1 summarises the visionary and leading MDM packages.

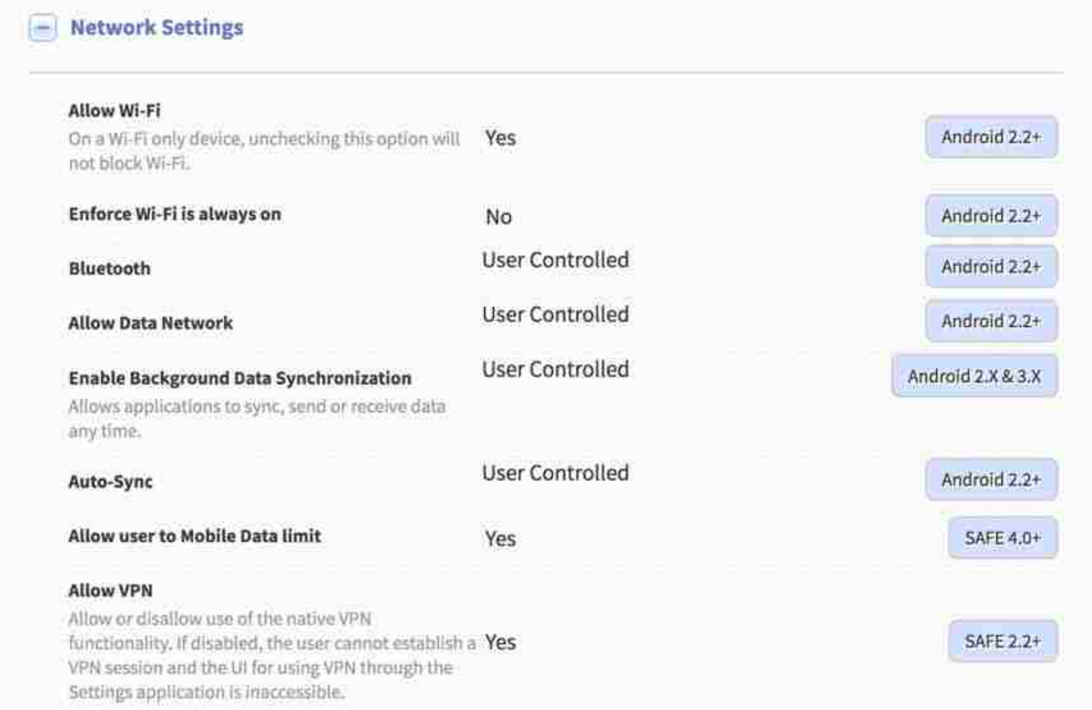


Figure 5.1: Policy settings in the MaaS360 MDM tool.

Whilst MDM tools can configure devices, the policies they enforce are less advanced than what can be written using a policy language such as AppPAL. The policies of an MDM tool are essentially granular check boxes (shown in Figure 5.1). An administrator can manually enable or disable features to configure devices for users.

5.1.3 Related BYOD Work

As well as commercial MDM products there are also research tools that are similar to MDM software. Martinelli et al.'s work creates a dynamic permissions manager, called UC-Droid. Their tool can alter what an app's Android permissions are at run time, based on policies [83]. The tool allows companies to reconfigure their apps depending on whether the employee is at work, in a secret lab, or working out-of-hours. These kinds of policies are more configurable than the geo-fenced based policies some MDM tools offer. Other work has looked at enforcing different policies based on what roles an employee holds [37]. The work allowed a company to verify the devices within their network and what servers and services they could reach. It also describes a mechanism for providing different users with different policies.

Armando et al. developed BYODroid as a tool for enforcing BYOD policies through a secure marketplace [11]. Their tool allows companies to distribute apps through a secure app store [12]. The store ensures apps meet policies through a static analysis and app rewriting to add dynamic enforcement. Their policies are low-level, based on ConSpec [5], and check the policy with respect to the Dalvik VM's state. Using their tool, they implemented the parts of a NATO Communications and Information Agency policy about personal networks and data management [10]. Their work shows how to check and enforce the app-specific sections of a BYOD policy using tools. They did not look at where the checks or policies come from, however.

Question	What are the idiomatic concerns in BYOD policies?
Input	Five different natural language BYOD policies from various sectors.
Output	The encodings of the policies into AppPAL. Analysis of policy contents.
Method	1. For each policy encode it as an AppPAL policy, taking care to capture the style and intent of the original policy. 2. Go through each encoded policy and ensure predicates have a common name where they capture similar decisions. 3. Compare AppPAL policies looking for areas of common concern and where common structures have been used to capture the policy.
Hypothesis	Different BYOD policies will have some common areas of concern and predicates. Some may be similar to the features associated with MDM tools, and others may be different. Comparing the AppPAL translations of the policies will help clarify the differences and similarities between the policies.

Table 5.2: Summary of method to identify idioms in BYOD policies.

5.2 Modelling BYOD policies

BYOD policies written in natural language can have ambiguities. Comparing policies with different styles from different sources is tricky: the relationships between different entities, and the exact decisions being made can become confused. To make the comparisons precise, and to identify common authorisation patterns in the polices we propose the method summarised in Table 5.2: encode each of the natural language policies in AppPAL and then use the more formal AppPAL version as the basis for examining the policies. We took care to use the same predicates between different policies in order to create a standard set of decisions between BYOD policies. We hypothesised that by doing this we would be able to make comparisons and identify common BYOD idioms between policies.

Each of the policies are split into a series of rules. A simple example is the following example from the Sirens company policy. The rule states that devices may get access to various company resources. For each resource we create an AppPAL assertion stating the device may access the resource.

Sirens: *Employees may use their mobile device to access the following company-owned resources:*

• *Email* • *Calendars* • *Contacts* • *Documents* • *Etc.*

```
'department' says Device:D canAccess('email').
'department' says Device:D canAccess('calendars').
'department' says Device:D canAccess('contacts').
'department' says Device:D canAccess('documents').
```

The NHS policy has a more complex example. Employees are not allowed to call non-domestic, or premium rate numbers on company-owned phones. An employee's manager, however, can make an exception. To write it in AppPAL we first describe the rule that bans international calls. We add a second rule stating that it is allowed if someone exempts it. A third lets the employee's manager make the exemption.

NHS: *All mobile devices will be configured for national access only. Premium/international calls will be barred. International call barring and roaming arrangements can be lifted for specific periods, to be stipulated on request, on approval of the relevant manager/budget holder.*

```
'nhs-trust' says Device canCall(TelephoneNumber:X)
  if Device isOwnedBy('nhs-trust'),
    X isNationalNumber, X isStandardRateNumber.

'nhs-trust' says Device canCall(TelephoneNumber:X)
  if Device isOwnedBy(Staff),
    Staff hasCallExemption.

'nhs-trust' says Manager can-say
  Staff hasCallExemption
  if Manager isManagerOf(Staff).
```

Table 5.3 shows a count of the different predicates used in the policies. As in Chapter 3, we use prefixes to distinguish different kinds of predicates. The use of each is also split by whether the predicate is a *decision* made by the policy (i.e. it exists in the *head* of a rule), or a *condition* for making that decision (i.e. it exists in the *body* of a rule). *Can* and *must* decisions feature in all policies, except *can* decisions in the Edinburgh policy due to its structure (as discussed in Section 5.1.1). This is expected. Access control decisions and reactions to events are both topics that MDM tools have focused on implementing. *Has* and *is* predicates are often used in the conditions. There are also decisions using them as well.

5.3 BYOD Idioms in AppPAL

We hypothesised, when translating the policies into AppPAL, that we might be able to identify idiomatic decisions for BYOD policies. These idioms would appear as common predicates, and structures in the AppPAL policies. Examining the policies there are some concern concerns shared between the different policies. Table 5.5 shows predicates used in multiple policies by our translation. The table lists predicates (and arguments) found in multiple policies

and in which policies they occurred. From it we can find the common decisions of concern to BYOD policies.

Acknowledgements, where the policy requests people acknowledge other policies, and predicates linking devices to owners feature in all policies. Most policies describe rules for when to enable and disable device features. Configuring device features is a found in many MDM packages, but tracking what a user agrees to is not seen in MDM packages. Only two of the five policies had rules limiting access to networks, servers, or access points. This is surprising as many MDM tools can control how devices and apps access networks. Users have privacy preferences about apps [82], but not all companies try to control what employees can install. Curated app stores and app blacklisting are common features to many MDM programs; but not all policies have rules about which apps to install. All policies talk about remotely wiping a device for security reasons (as shown in Section 5.1.1). Most MDM tools offer this: they let admins remotely erase devices (Table 5.1).

Two particular idioms occur in many policies: acknowledgements and delegation. So far MDM tools and research have focused on implementing restrictions on apps and devices [67, 13, 83]. These controls are a vital aspect of BYOD policies and all five of the policies we looked at had rules that described restrictions (Table 5.5). Every policy also had rules that required employees acknowledgements, however. Only the (configuration focused) SANS policy had more rules that required restrictions than acknowledgements. All the policies had more rules featuring delegation relationships than functionality restrictions. Restricting device functionality is and important: but other aspects of BYOD policies are also worth attention.

5.3.1 Delegation and Roles Within Policies

Delegation is an important part of the policies. Each of the policies describes through rules how separate entities are responsible for making some decisions. These rules could be a delegation to an employee's manager to authorise a decision (as in the NHS policy). It could be to technical staff to decide what apps are part of a standard install (as in the sirens and SANS policies).

When translating the policies, the author of the policy is the primary speaker of the policy's rules. This is typically the company; except in the HiMSS case

Policy	Decision Can	Decision Must	Decision Has	Decision Is	Condition Can	Condition Must	Condition Has	Condition Is
SANS	35% (26)	29% (22)	9 % (8)	27% (20)	2% (2)	2% (2)	8% (9)	87% (82)
HiMSS	21% (6)	41% (12)	31% (9)	7 % (2)	0	0	13% (3)	87% (20)
NHS	19% (13)	26% (18)	33% (23)	23% (16)	2% (2)	0	19% (20)	83% (83)
Sirens	27% (12)	45% (20)	11% (5)	16% (7)	2% (1)	7% (4)	2% (1)	89% (50)
Edinburgh	0	18% (3)	82% (9)	0	7% (2)	7% (2)	50% (15)	37% (13)

Table 5.3: Counts of predicate-types in each policy.

	Authorities	Primary Authority	Technical Authority	User Authority
SANS	10	company	it-department	user
HiMSS	3	user	xyz-health-system	department
NHS	11	nhs-trust	it-department	employee
Edinburgh	2	records-management		employee

Table 5.4: Summary of different authorities in BYOD policies.

where it is the user (Table 5.4). All the policies describe multiple entities that might make statements and delegate.

Some policies have more authorities than others (Table 5.4). The NHS policy has various managers that approve decisions for their staff. There are groups that make decisions for the clinical and business halves of the NHS. If a clinical user wishes to use an app with a patient they must seek approval from two policy groups, as well as their line manager.

Others make less use of different authorities. In the Edinburgh policy, the records-management office states how to configure a low or high risk device. There is no delegation to others to further specify aspects of the policy.

Whilst delegation of responsibilities is an important part of BYOD policies, many MDM tools seem to largely to ignore it, however. These tools instead allow IT staff to set fixed policies and push them to devices. No further requesting of information is typically needed or required.

When a policy decision requires input from a third-party, we use delegation. For example, an employee's manager must authorise an app install. The SecPAL *can-say* statement is the basis for a delegation. We can ask the HR department to

Predicate	SANS	HiMSS	NHS	Sirens	Edinburgh
person mustAcknowledge*(policy)*	✓	✓	✓	✓	✓
person hasAcknowledged*(policy)*	✓	✓	✓	✓	✓
device isOwnedBy*(person)*	✓	✓	✓	✓	✓
thing isDevice	✓	✓	✓	✓	✓
device mustDisable*(feature)*	✓		✓	✓	✓
device mustWipe		✓	✓	✓	✓
device isLost	✓	✓	✓	✓	
thing isEmployee	✓		✓	✓	✓
thing isApp	✓	✓	✓	✓	
device isActivated	✓	✓	✓		✓
device mustEnable*(feature)*	✓	✓		✓	
device isEncrypted	✓		✓		✓
device hasFeature*(feature)*	✓		✓		✓
device hasMet*(policy)*	✓		✓		✓
person canMonitor*(device)*	✓		✓	✓	
person mustInform*(person)*	✓		✓		
thing isTelephoneNumber	✓		✓		
thing isString			✓	✓	
thing isSecurityLevel	✓	✓			
app isInstallable	✓		✓		
thing isFeature	✓			✓	
thing isData	✓			✓	
application isApprovedFor*(person)*		✓	✓		
application isApproved	✓	✓			
person hasDevice*(device)*		✓	✓		
person hasDepartment*(department)*		✓			✓
person canUse*(device)*	✓		✓		
device canStore*(file)*	✓	✓			
device canInstall*(app)*	✓		✓		
device canConnectToServer*(server)*	✓		✓		
device canConnectToNetwork*(network)*	✓			✓	
device canConnectToAP*(access-point)*	✓	✓			
device canCall*(number)*	✓		✓		
device canBackupTo*(server)*		✓			✓

Table 5.5: Occurrences of predicates common to multiple policies.

state who is someone's manager. When we delegate, we can add conditionals to the can-say statement that enforces any relationship between the delegating and delegated parties.

```
'company' says 'hr-department' can-say
  Employee:E hasManager(Employee:M).
'company' says Manager can-say
  Employee canInstall(App:A)
  if Employee hasManager(Manager).
```

5.3.2 Acknowledgement

All the policies we looked at require their subjects be aware of and acknowledge certain rules or policies. These include acknowledging the company may do certain actions. Requiring subjects to acknowledge and agree to follow other policies is interesting as these are not necessarily technical restrictions on employee behaviour. An administrator cannot configure a device so that a user is aware of an ethical policy. Rather, they must trust that when the user says they've read the policy, they actually did so.

In some cases this is used by companies to justify their actions. A user cannot complain about a company's actions if they were notified in advance. For example the NHS and HiMSS policies state that the organisation will wipe devices remotely to protect confidential information if a user loses their device. Both policies also say that employees would lose personal information if they had it on the device and the company needed to erase it. In the case of the HiMSS policy, the user agrees not to sue the company if this happens.

HiMSS: *I agree to hold XYZ Health System harmless for any loss relating to the administration of PDA/Smartphone connectivity to XYZ Health System systems including, but not limited to, loss of personal information stored on a PDA/Smartphone due to data deletion done to protect sensitive information related to XYZ Health System, its patients, members or partners.*

```
'xyz-health-system' says
  'user' mustAcknowledge('data-loss-policy').
```

NHS: *Individuals who have personal data of any kind stored on a corporately issued mobile device must be aware that in the event of loss of the device the above data wipe will include removal of all personal data.*

```
'nhs-trust' says Staff:S can-say
  S hasAcknowledged('data-loss-policy').
'nhs-trust' says
  Staff:S mustAcknowledge('data-loss-policy').
```

The SANS, NHS and the Sirens company policies use acknowledgements to link to other sets of rules that employees should follow, such as the *acceptable-use* policy in the SANS example bellow. These policies are not further specified and, in the case of an acceptable use policy, are hard to enforce through other means. The SANS policy requires that all employees follow an email security, acceptable use, and an eCommerce-security policy. The Sirens policy expects an employee to use their devices ethically and abide by an acceptable use policy.

SANS: *Users MUST agree to the email security/acceptable use policy and eventually to the eCommerce security policy.*

```
'company' says
  Employee:U mustAcknowledge('email-security').
'company' says
  Employee:U mustAcknowledge('acceptable-use').
'company' says
  Employee:U mustAcknowledge('ecommerce-security').
```

Sirens: *The employee is expected to use his or her devices in an ethical manner at all times and adhere to the company's acceptable use policy.*

```
'department' says
  Employee:E mustAcknowledge('acceptable-use').
```

We use acknowledgements when employees should be aware of (usually separate) rules. The company may not have wish to enforce these separate rules automatically, however. A company may have an ethical policy that says employees should not use devices for criminal purposes. The company is not interested in, or capable of, defining what is criminal. They trust their

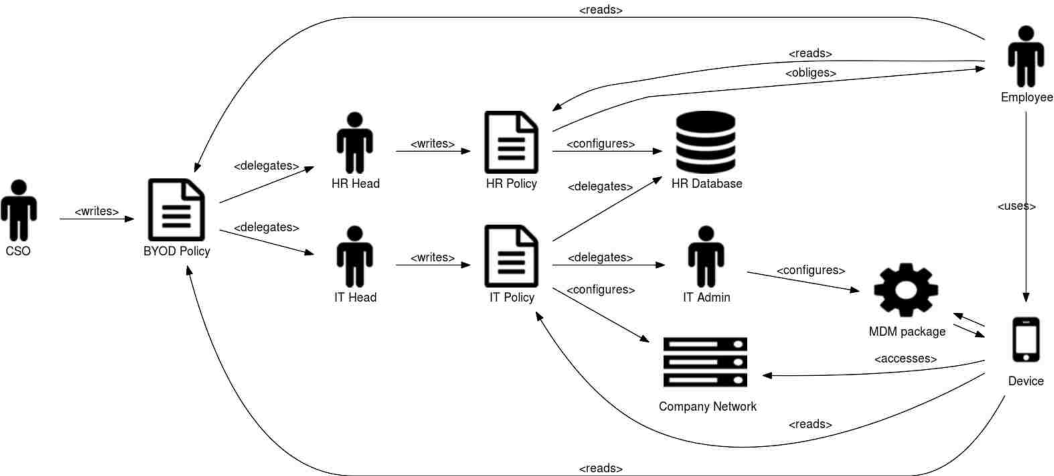


Figure 5.2: Interactions in a company with BYOD security policies.

employees to make the right decision. They trust their employees to know of the rules.

To write these in AppPAL, a policy author creates two rules. The first states their employees must have acknowledged the policy. The second delegates the acceptance of the policy to the employee themselves.

```
'company' says Employee:E mustAcknowledge('policy').
'company' says Employee:E can-say
  E hasAcknowledged('policy').
```

5.4 Enforcing a BYOD policy with AppPAL

Our work has, so far, looked at the BYOD policy contents. We have highlighted the concerns and delegation relationships as they are not handled well by existing MDM tools. The policies also have rules that the MDM tools do manage well: configuring, provisioning and managing devices. The MDM tools fail, however, to give a means to control *how* they to configure themselves. Typically, an IT administrator configures them manually.

AppPAL exists as a tool for checking whether a policy is contains sufficient facts and rules to decide whether a given statement is true or not. It is reasonable to ask what would a company have to do to *enforce* their mobile device policies. A company could have a structure such as in Figure 5.2: the Chief Security Officer sets the BYOD policy, which delegates to IT and HR to further specify parts of the policy. HR obliges employees to read all the policies put out

and follow them (by HR). The IT department configure MDM software on employee's devices to ensure the devices meet their policies.

AppPAL offers a company the means to describe the policy precisely and check if it satisfied. To enforce the policy the company would need to add actions on top of these checks. If AppPAL checks a policy and finds a user must install an app or disable a feature on their phone, then the company might want to configure their MDM software to do something to correct it. If AppPAL found an employee had not met the obligation to sign a contract, then the company might want to email a reminder to the employee.

AppPAL doesn't remove the need for MDM software. Much of an MDM tool could be re-implemented as part of AppPAL: but this is an unnecessary duplication of work. Rather, AppPAL should be used to configure and with other existing tools. An MDM package could enforce a password policy, and enable remote wipe. Google's *For Work* tools can enforce access control policies for company's documents, as can Microsoft's Office Suite. The settings on an Android app control what permissions an app can have. MDM tools give a means to control how a mobile device behaves, and what employees can do with a device. Alone, however, they do not give enough to fully enforce a BYOD policy as all the BYOD policies we looked at had more than just device configuration.

AppPAL works as a glue between existing tools. It tells an administrator what tools are used to check a policy. If the tool's configuration can be automated through AppPAL (perhaps by enabling certain settings), then so much the better. In this respect AppPAL is similar to a configuration language. Giving employees a means to make AppPAL statements (such as a dialog box that says *"to acknowledge the following policy type your password."*) lets us track what people have done and make decisions on the basis of their actions. Configuring and selecting the tools lets us pick which checks we enforce and when.

Chapter 6

Future Work

We have described the policies of the mobile ecosystem and how to use AppPAL to describe them. We have argued that AppPAL is a good language for describing these policies, however there are also areas where AppPAL could be improved to further to describe more kinds of policies and to aid policy authors, as well as examining further aspects of the mobile ecosystem. This chapter suggests areas for future research and some starting ideas as to how they might expand our knowledge of mobile ecosystems and the trust relationships within them.

6.1 Probable SecPAL

The SecPAL authorization language, and the AppPAL instantiation, let policy authors to make use of static analysis tools to make decisions, and allow principals to make statements about apps through delegation. When these decisions are made, they are made with certainty. If a principal says an app is safe to access the network, then we believe that that principal definitely believes the app is safe on a network. When a static analysis tool finds that an app isn't malware, then we believe that app to not be malware. This isn't realistic. Static analysis tools can produce false results. A principal might be merely fairly confident that an app can access the network safely but not absolutely certain.

With current authorization languages you cannot quantify the belief a principal has in any statement. A principal cannot say how *probable* they think any statement is.

6.1.1 Examples of Probability

SecPAL was designed to make access control decisions. The decision whether to install allow a user access to a file or not is a binary one: either they can access it or they cannot. Similarly, the decision process for these decisions is also binary: a user is either logged in or not, a network address is either in the network or outside it, someone can act as someone else's manager or they can not. Not all decisions are binary however.

As an example of a probabilistic policy consider a user, Alice, who has a policy to only install safe apps, ones she knows are not malware. She could use an AV program, but these tools are not infallible as they can change their judgment about apps over time. Others AV tools are heuristic based and may only be capable of judging an app based on sufficiently many malware indicators being present. Instead of using one AV program, suppose that Alice opts to use *VirusTotal*; a web service for running files through multiple AV programs. Even for known malware samples VirusTotal rarely gives absolute answers instead giving the number of AV programs that flagged it (Figure 6.1). Alice acknowledges this and writes her policy accordingly:

```
'alice' says App:A isInstallable
  if A isSafe
  with probability at least 0.75.

'alice' says 'virustotal' can-say
  App:A isSafe

'virustotal' says 'com.good.app' isSafe
  with probability 0.95.

'virustotal' says 'com.malicious.app' isSafe
  with probability 0.26.
```

She can install the `com.good.app`, since the probability it is good is greater than her threshold but the `com.malicious.app` fails her test and is uninstallable.

In the first example VirusTotal gave Alice a value for how probable it was the app was safe: but what if Alice has to decide this value herself? An example of this is app recommendations. Alice only wants to install apps that are really good. She has two friends, Bob and Charlie, who offer her reviews of a game. To complicate matters further, whilst she trusts Bob utterly, she is a little less

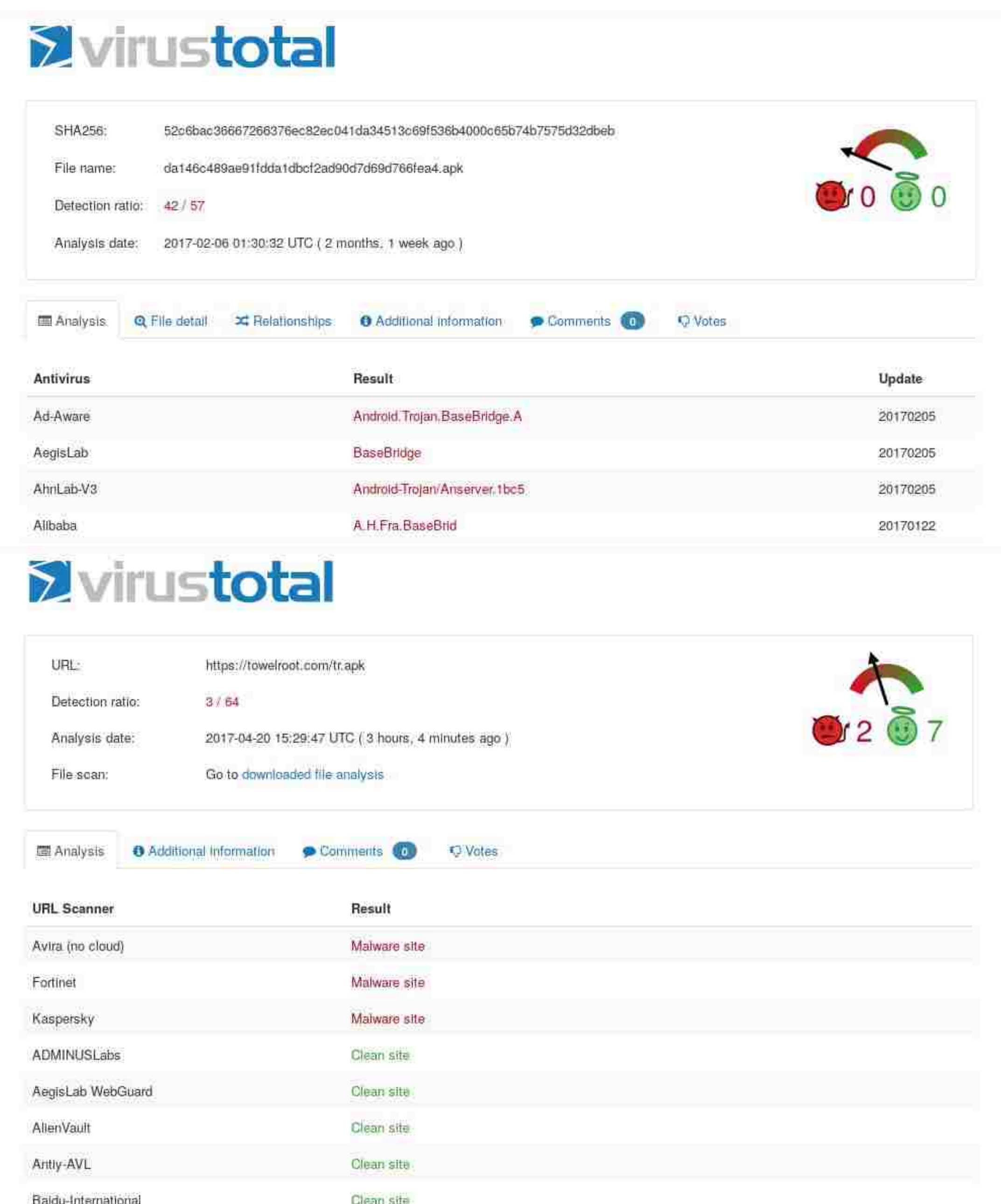


Figure 6.1: VirusTotal results for 57 antivirus packages scanning a sample of the BaseBridge Android malware, and 64 antivirus packages analyzing a download of the (relatively safe) Towelroot rooting app. The tools do not agree on the results.

trusting of Charlie.

```
'alice' says App:A isInstallable
  if A isGood
  with probability

'bob' says 'com.rovio.angrybirds' isGood
  with probability 0.8.

'charlie' says 'com.rovio.angrybirds' isGood
  with probability 0.9.

'alice' says 'bob' can-say App:A isGood.

'alice' says 'charlie' can-say App:A isGood
  with probability 0.9.
```

How should she combine the information to get an overall rating of the game? If Alice wants to only install apps that are both good and safe how should she trade off the probability against each other? Is she willing to install a more dangerous app if it is highly reviewed?

Various papers have proposed probabilistic variants of Datalog [49] or explored the semantics of probabilistic logics [61]. Role-based access control languages have incorporated ideas about risk into their schemes [72, 42, 100], which is a similar notion to probability and trust. These schemes do not seem to deal with delegation in the same manner as SecPAL however so incorporating similar ideas here may be interesting and allow SecPAL and AppPAL greater expressiveness.

Whilst some work went into developing the ideas behind a probable SecPAL variant, we did not finish this work because there are several hard problems surrounding it. In particular, it is far from clear how to combine probabilistic statements in a consistent, and sound manner. Several choices are possible but more research is needed to understand the right way to do it and how to develop Probable SecPAL into a full, and useful language.

6.1.2 Guarantees for Probable SecPAL

In Appendix B we suggest how we might implement Probable SecPAL by modifying Becker et al.'s original design. Regardless of how we implement it, we

should consider carefully how we might *combine* information and probabilities. Combining probabilities of events when you cannot guarantee independence is hard, and the same is true of probability. Strategies such as taking the product, minimum of combined statements may be too simple. We suggest the following properties should hold no matter what combination mechanism is used:

1. If all facts are completely probable, then evaluation should be equivalent to standard SecPAL.
2. If any fact is completely improbable, then it should be equivalent to the statement not existing in the assertion context.
3. No derived statement should be more probable than the conditions used to derive it.

Rule 1 ensures that any probability additions do not start to produce different results to standard SecPAL. We wish to be able to reason about scenarios where we have partial probability, but this addition shouldn't change our ability to reason when each speaker has total belief in their assertions. If everything is completely probable, then the reasoning should be equivalent to SecPAL where the perfect probability is assumed. Similarly, if a speaker believes a fact to be completely improbable, then Rule 2 ensures that fact is used to derive other facts. SecPAL operates under a *closed world assumption*, that is that the assertion context contains (or at least can derive) *all* known facts. If a statement is missing, then it is false. If a speaker believes a statement to be perfectly improbable then it should be equivalent to falsehood in standard SecPAL and should not be used further.

Rule 3 applies to assertions with a conditional part (i.e. an `if`). If this rule were not the case, we might be able to grow the probability of a fact by applying a rule that contained its own derived fact in its condition. For example, consider the following rule:

```
'x' says 'y' p
  if 'y' p,
     'z' q.
```

Say we know already that by applying this rule the probability that `'x' says 'y' p` will be greater than the probability of the conditionals that `'x' says 'y' p` and `'x' says 'z' q`. Since the decision that `'x' says 'y' p` is also part of

$F(\phi)$	$\Rightarrow$	At sometime in the future ϕ is true.
$P(\phi)$	$\Rightarrow$	At sometime in the past ϕ was true.
$G(\phi)$	$\Rightarrow$	ϕ will always be true.
$H(\phi)$	$\Rightarrow$	ϕ was always true.

Table 6.1: Summary of temporal operators from Prior.

the conditions, we could repeatedly reapply this rule and raise the probability arbitrarily.

6.2 Patterns with Predicates

We described in Chapter 3 how we split AppPAL predicates into four types: can, has, is and must. Using the four predicate types we can start to build relationships between them. We also showed how a policy author could check an obligation was completed when describing *must* predicates. Whilst a policy author could write these rules by hand, AppPAL has tools to automate creating rules like this.

As well as automating creating the policy, we could also start to check other properties to check the policy is well-formed. Using the example of installing an app; if *Alice says Bob must install an app*, then it implies that there should be a rule where *Alice says Bob has installed that app*. We might also expect there to be a rule that *Alice says Bob can install the app*.

This can be taken further: if in one version of the policy Alice *can* do something we might expect to see in a later version of the policy assertions where Alice *has* done something. We could generalise this and create a new rule that if someone has done something, then there must have been a point where the policy said they could do it.

Using temporal operators, such as those established by Prior [92] (Table 6.1), we can start to write rules expressing the relationship between different predicates. Taking the earlier example, we might wish to add a rule that if Alice believes Bob has done something ϕ, then she must have said he could do that thing in the past.

$$\frac{AC, D \models \texttt{A says B has } \phi.}{P\left(\exists AC' \text{ s.t. } AC', D' \models \texttt{A says B can } \phi.\right)}$$

This doesn't mean that if Alice *can* do something then this will always lead to a statement where she *has* done it in the future, but it does imply that if she has done something, then the action must have been permitted in the past.

Using this structure we could add a rule to AppPAL that all decisions from the past, will hold in the future. This might enable us to reduce the size of the assertion context as once we have a statement that someone *has* we can remove the respective *can* assertions. Alternatively if we prove the rule is false, i.e. at no point in the past did Alice permit Bob's action then we can detect something has gone wrong.

$$\frac{AC, D \models \texttt{A says B has } \phi. \quad \neg P\left(\exists AC' \text{ s.t. } AC', D' \models \texttt{A says B can } \phi.\right)}{\bot}$$

An AppPAL interpreter might implement this by requiring the *can* statement be made before any equivalent *has* statement is made. Looking at a trace from an instance of an AppPAL interpreter with one AC (though which might import or remove assertions over time), the following trace would be valid, but if the statement in red was removed it would be rejected.

⋮	⋮	
At T_i:	$AC_i, D \models \texttt{A says B can } \phi.$	
⋮	⋮	
At T_j:	$AC_j, D \models \texttt{A says B has } \phi.$	(j > i)
⋮	⋮	

Interesting examples are not just limited to *can* and *has*: if Alice says Bob *must* do some ϕ, then we would expect that at some point in the future (if not immediately) that Alice will say that Bob *can* complete the obligation to ϕ. Furthermore, we would also expect Alice to acknowledge that at some future point Bob *has* done ϕ as she required.

$$\frac{F\left(\exists AC' \text{ s.t. } AC', D' \models \texttt{A says B can } \phi.\right) \quad AC, D \models \texttt{A says B must } \phi.}{F\left(\exists AC' \text{ s.t. } AC', D' \models \texttt{A says B has } \phi.\right)}$$

Using this interpretation of AppPAL's rules the following for an interpreter the following trace is valid; but if either statement in red was removed the interpreter would report an error.

$$
\begin{array}{lll}
\vdots & \vdots & \\
\text{At } T_i\text{:} & AC_i, D \models \texttt{A says B must } \phi. & \\
\vdots & \vdots & \\
\text{At } T_j\text{:} & AC_i, D \models \texttt{A says B can } \phi. & \\
\vdots & \vdots & \\
\text{At } T_k\text{:} & AC_j, D \models \texttt{A says B has } \phi. & (\text{k} > \text{i}, \text{k} > \text{j}) \\
\vdots & \vdots & \\
\end{array}
$$

Further investigation into the relationships between predicates, and their semantics, might allow for interesting auditing possibilities with AppPAL and extend the language further.

6.3 AppPAL MDM

BYOD is another area where it would be interesting to see how AppPAL might be used in practice. AppPAL is designed to separate the policy specification from its implementation. By combining an AppPAL policy engine with an MDM package we could dynamically configure the MDM software to enforce a BYOD policy. This would allow for greater customization as we could extend the MDM tool to support contextual and policy driven controls rather than the low-level simple ones that are the only controls many tools support. More advanced MDM tools, such as Armando et al.'s *meta-market* [12], can enforce more complex policies but cannot distinguish between different contexts. A user who uses an app for both work and home may wish to run an app unencumbered and with different accounts when at home compared to at the office. Extending a tool such as this with AppPAL would allow contextual as well as policy controls into apps.

Trialing an implementation in a company and working with the IT departments and policy authors to better understand their needs would allow us to tailor AppPAL further to implementing BYOD policies rather than just modeling them.

6.4 Usability Study

AppPAL is designed to be readable, but we might guess that most non-technical users would struggle with writing a formal policy (as they do with many other technical tasks). Exploring how general users might use policies, either through a subscription model where they *subscribe* to a policy written by a more advanced user, or through a graphical interface that helps them build their own policy, would help us understand how (and in fact whether at all) users would take advantage of better app controls.

Looking solely at the language a user study with AppPAL should seek to answer at least three questions looking at a range of users of varying technical skill and comparing to results from other languages (such as XACML):

Can users understand a policy written in AppPAL? Becker designed SecPAL to be readable but didn't trial the language with users. Specifically SecPAL was designed to be less verbose than XACML, to abstract away the logic-based syntax of the RT family (which some administrators found intimidating), and to offer a simple succinct syntax based on natural language. Measuring users' comprehension of AppPAL would test whether the syntax offers an advantage over other languages.

Can users predict the decision made by AppPAL? As well as being readable SecPAL was designed to have intuitive semantics. We would hope that a user should be able to follow and understand the decision process made by AppPAL, but we should test it with users to be sure. A key requirement for a policy language should be how predictable it is as we do not want the language to make surprising decisions (for its users), and allow or deny actions that users intuitively believe should not be permitted given a policy. Testing whether users can predict decisions given a policy and whether they can explain, in high-level terms, the decision process would give some measure of the language's intuitiveness.

Can users modify or create an AppPAL policy? If AppPAL is comprehensible and predictable we would hope that a user could tweak or even create a policy to meet their own requirements. SecPAL was designed to be expressive, but we should test whether users are capable of expressing the policies they want using it.

Chapter 7

Conclusion

This thesis looked at capturing the security policies of the mobile ecosystem precisely. We have tried to capture the trust relationships within the policies using the AppPAL policy language we instantiated from SecPAL. This allowed us to make precise comparisons between different policies, and study the way policies were used in practice rigorously.

Our work contributes to the literature by showing how to capture the security policies of the mobile ecosystem, and how to use the formal policies as the basis for comparisons and reasoning. Whilst AppPAL does not have any new semantic language features, our work is novel as it shows the application of policy languages to a new domain.

One benefit of AppPAL is that it lets us separate policy specification from implementation. We can describe the policies of the mobile ecosystem at a higher level than other tools, delegating to the low-level tools when we need and when we want them to do their analysis. This makes our policies independent of any particular tool, and lets us abstract the checking process away from the reasons the policy requires we do the checks.

In some policies, *who* has performed a check is more important to the policy than *what* check they actually did. Existing research has looked at ways of enforcing policies mechanistically. Sometimes, however, just trusting the subject to follow them on their own, however, is enough. Capturing these trust relationships lets us see precisely how a policy is satisfied.

Our work is not without its limitations. There are many policies in the mobile ecosystem that we didn't fully look at; these include policies such as the differences between OSs and the differences between store contents. We have

shown that by using AppPAL we can gain a better understanding of the policies, however. This includes the policy's implications and the trust relationships between companies, people and tools they use.

In his Turing award lecture *on trusting trust* Ken Thompson asked a question [106]:

> *"To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software."*

He concluded:

> *"The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code."*

The same question could be asked of the policies in the mobile ecosystem. Thompson's conclusion is still true: there is no way to trust that any app you download is safe or good. There is no way to check employees are not circumventing a BYOD policy cleverly. It is hard for users to create all their own apps and OSs. Few companies can only run with only one employee. We can't trust people follow our policies. But we can understand the trust relationships within them. If we understand the trust relationships, we can decide what risks we take, and with whom.

Appendix A

Translated BYOD Policies

A.1 NHS

Torbay and Southern Devon Health and Care Mobile Devices Policy

AppPAL Translation

A translation of the requirements in this document into AppPAL is presented in this column. Not all policies are translatable (or contain an actual requirement that should be enforced), but where they are the rule is given next to the informal policy text.

1. *Glossary of Terms*

Mobile device: a mobile device is a device that enables functionality away from the main base of work.

Mobile phone: a mobile phone is a device that allows the ability to make telephone calls and send and receive text messages without the need of a physical connection to the telephone network.

Smartphone: a smartphone provides the same functionality as a mobile phone with the additional ability of being able to send and receive emails and enabling the use of general purpose applications (apps).

Tablet: a tablet computer, or simply tablet, is a mobile computer with display, circuitry and battery contained within a single device.

Smart tablet: a smart tablet comprises the ability of a tablet with the functionality of a smartphone with the exception of not being able to send or receive phone calls or text messages.

Supported and non-supported devices: a supported device is one that the Trust issues and will provide technical support for. A non-supported device does not fall within this remit.

This policy contains many requirements which depend on an employee having agreed to various conditions. We can simplify the policies if we require employees to agree to these conditions up front.

```
'nhs-trust' says Staff:S mustAcknowledged('acceptable-use').
'nhs-trust' says Staff:S mustAcknowledged('charger-policy').
'nhs-trust' says Staff:S mustAcknowledged('charging-policy').
'nhs-trust' says Staff:S mustAcknowledged('confidentiality-guidelines').
'nhs-trust' says Staff:S mustAcknowledged('damage-policy').
'nhs-trust' says Staff:S mustAcknowledged('data-loss-policy').
'nhs-trust' says Staff:S mustAcknowledged('driving-policy').
'nhs-trust' says Staff:S mustAcknowledged('monthly-fee').
'nhs-trust' says Staff:S mustAcknowledged('personal-liability').
'nhs-trust' says Staff:S mustAcknowledged('public-usage-policy').
'nhs-trust' says Staff:S mustAcknowledged('usage-policy').
'nhs-trust' says Staff:S mustAcknowledged('work-communication-policy').
```

2. *Eligibility Criteria*

2.1.

The Trust is committed to flexible working and ensuring that adequate communication facilities are available to its staff in order for them to carry out their normal daily duties. Devices will be allocated according to the criteria below and depending upon both the person's role and the location in which they normally work.

The criteria for a mobile device is as follows (all three must be met):

```
'nhs-trust' says Staff isNeeding('Mobile')
  if Staff isLoneWorking,
    Staff isOutOfHoursWorking,
    Staff isOutOfOfficeWorking.
```

- Staff whose work entails predominantly lone working in the community
- Emergency out of hours staff including any staff on the on-call rota
- Staff who spend a significant amount of time out of the office and are required to be contactable during this period

2.2.

In order to qualify for a smart device a member of staff must first qualify for the above basic mobile device criteria. However, where it has been agreed that a mobile device with the ability to access the internet can be demonstrated to provide better and / or cost effective patient care (for example increased quality of work, improved productivity), a smart device may be considered. At least two of the following criteria must be met in order to apply for a smart device:

- Staff whose work regularly requires the use of email whilst lone working in the community / working out of the office.
- Staff whose work regularly requires access to their calendar whilst lone working in the community / working out of the office.
- Staff whose work requires internet access whilst lone working in the community / working out of the office.

```
'nhs-trust' says Staff isNeeding('SmartMobile')
  if Staff isNeeding('mobile'),
     Staff hasMet('smart-mobile-requirements').

'nhs-trust' says Staff hasMet('smart-mobile-requirements')
  if Staff isNeeding('oooemail'),
     Staff isNeeding('ooocalendar').

'nhs-trust' says Staff hasMet('smart-mobile-requirements')
  if Staff isNeeding('oooemail'),
     Staff isNeeding('ooointernet').

'nhs-trust' says Staff hasMet('smart-mobile-requirements')
  if Staff isNeeding('ooocalendar'),
     Staff isNeeding('ooointernet').
```

2.3.

If the Trust deems it is not necessary for a member of staff to have a mobile device to discharge their duties and has decided not to issue a device, that member of staff should not use a personal mobile device for Trust business. If an individual does use a personal mobile device for Trust business they do so entirely at their own risk and own cost. There are some circumstances in which the Trust prohibits the use of personal devices for Trust business and this includes any circumstances where personal identifiable data is stored on that device. Please refer to the confidentiality staff code of practice for further advice and guidance.

```
'nhs-trust' says User canUse(Device)
  if Device isOwnedBy(User),
     User hasAcknowledged('personal-liability'),
     User hasAcknowledged('confidentiality-guidelines').


'nhs-trust' says User canUse(Device)
  if Device isOwnedBy('nhs-trust'),
     User hasAppliedFor('phone', Form),
     Form isAuthorized.
```

2.4.

Upon receipt of the completed application form, allocation will also need to be authorised by the relevant Assistant Director and the Director of Finance.

```
'nhs-trust' says Staff hasAppliedFor('phone', Form)
  if Staff hasSubmitted(Form),
     Form isReceived.

'nhs-trust' says Form isAuthorized
  if Staff hasAppliedFor('phone', Form),
     FinanceDirector:F hasApproved(Form),
     AssistantDirector:D hasApproved(Form),
     D isManagerOf(Staff).
```

3. *Mobile Device User Roles and Responsibilities*

3.1. The corporately issued mobile device is the property of the Trust and as such it is a requirement that staff must take good care of it. Staff should take all reasonable steps to ensure that it is not damaged, lost or stolen.

```
'nhs-trust' says Employee:S can-say
  S hasAcknowledged('damage-policy').
```

3.2. In receiving a mobile device from the Trust, the individual receiving and using the device accepts that the device can be used to communicate through all corporate channels including voice calls, emails and texts and where appropriate enabled web applications, during working hours.

```
'nhs−trust' says Employee:S can−say
  S hasAcknowledged('work−communication−policy').
```

3.3.

There are circumstances in which members of staff may use the corporately issued mobile device for personal use. These are described in section 4 below. However, any personal data that is stored on the device will be covered by the following sections.

3.4.

In the event that a member of staff intends to use a corporately issued mobile device for personal use, a declaration of use form must be completed in which the user declares their intention to use the device for personal use. In this instance a minimum of 5 per month will be deducted from their salary towards the cost of personal calls made from the device. If the personal usage goes above the 5 per month the individual will be asked to pay the difference and in some circumstances the device may be withdrawn.

```
'nhs−trust' says Staff can−say
  Device isPersonalUse
  if Device isOwnedBy(Staff).
```

3.5.

It is the responsibility of staff to ensure that mobile devices are kept safe and secure. Any losses, damage or misuse should be reported immediately to the IT Department in order for the device to be disabled. If the device is stolen, staff will be expected to report the theft to the police and obtain a log number. An incident form should also be completed on Datix. Subsequently, if the device is found it can then be re-enabled by the IT Department, who should be informed immediately.

```
'it−department' says Staff can−say
  Device isLost
  if Device isOwnedBy(Staff).

'nhs−trust' says 'it−department' can−say
  Device isActivated.
```

3.6.

Guidance on the use of smart devices with NHSmail can be found on iCare. Once the smart device has been connected to NHSmail it is the responsibility of the device holder to ensure that only current work devices are linked to their NHS mail account.

```
'nhs−trust' says Staff can−say
  Device isLinkedTo(MailAddr)
  if Device isOwnedBy(Staff),
     Staff hasEmail(MailAddr),
     MailAddr isNHSMailAddr.
```

3.7.

In the event of the device getting lost, stolen, misplaced or updated, it is the responsibility of the device holder to manage the ‘remote wipe data’ through NHSmail. IT support is available if needed. Remote wiping is done by following these steps:

```
   'nhs−trust' says 'nhsmail' can−say inf
  Device mustWipe.

'nhs−trust' says Staff can−say Device mustWipe
  if Device isOwnedBy(Staff),
     Device isLost.
```

- Log in to NHSmail at `www.nhs.net`

3.8.

Individuals who have personal data of any kind stored on a corporately issued mobile device must be aware that in the event of loss of the device the above data wipe will include removal of all personal data.

```
'nhs-trust' says Employee:S can-say
  S hasAcknowledged('data-loss-policy').
```

3.9.

On connecting any Smart device to NHSmail, a minimum level of encryption is enforced. This will automatically apply a pin number or password. s

```
'nhs-mail' says Device canConnectToServer('nhs-mail')
  if Device hasFeature('encryption').
```

3.10.

It is the responsibility of the device user to ensure that the pedin number or password is kept up to date, remembered and kept secure at all times.

```
'nhs-trust' says Staff can-say
  Device mustUpdatePassword
  if Device isOwnedBy(Staff).
```

3.11.

It is the responsibility of the device user to keep the batteries fully charged and for the device to be kept switched on during working hours.

```
'nhs-trust' says Employee:S can-say
  S hasAcknowledged('charging-policy').
```

3.12.

It is the responsibility of the device user to ensure that mobile device chargers are only used for charging the correct devices. Mobile device chargers should only be plugged in for the duration of charging the device. Mobile device chargers left plugged in are a potential fire risk when not charging a device. When not in use, chargers should be disconnected and stored appropriately.

```
'nhs-trust' says Employee:S can-say
  S hasAcknowledged('charger-policy').
```

3.13.

Members of staff who have corporately issued mobile devices should remember to:

- Ensure they have their device with them when away from their office base.
- Ensure the device is switched on and they are able to receive calls, text messages and emails, where appropriate.
- Regularly check their device, particularly if it has been switched off for a period of time or if they have been in a black spot.

```
'nhs-trust' says Employee:S can-say
  S hasAcknowledged('usage-policy').
```

3.14.

The application process for allocating a mobile device to a member of staff requires the completion of an *electronic application form by the line manager*[1] via iCare. Managers must ensure that there is a clearly demonstrable business requirement for the device.

```
'nhs-trust' says LineManager:X can-say
  Staff hasMet('business-requirement')
  if X isManagerOf(Staff).
```

[1]See 2.4.

3.15.

When issued with a mobile device, members of staff will be asked to read this policy and will be required to complete the declaration of use form (Appendix A), which will be retained on the employee's personal file.

3.16.

The IT Department will monitor the device usage for excessive use and will bring any issues to the attention of the staff member and their manager.[2]

```
'nhs-trust' says 'it-department' canMonitor(Device)
  if Device isOwnedBy('nhs-trust').
```

[2] This is duplicated and extended by 4.4.4..

3.17.

The mobile device is intended for the exclusive use of the member of staff to whom it is issued. It should not be loaned or shared with anyone else including family members, friends or other members of staff. The use of this device will be monitored and any misuse could result in disciplinary action. The sim-card issued with the mobile device must be used only with corporate devices and must not be used with personally owned equipment unless otherwise authorised through the immediate line manager and the IT Department.

```
'nhs-trust' says Staff canUse(Device)
  if Device isOwnedBy(Staff).
```

3.18.

If the device is longer required or if it has been passed on to a colleague, the Line Manager and IT Department must be informed via the following electronic form.

```
'nhs-trust' says Staff can-say
  Device isNoLongerRequired
  if Device isOwnedBy(Staff).

'nhs-trust' says Employee:S can-say
  Device hasPassedOnTo(Employee:T)
  if Device isOwnedBy(S),
     S hasAppliedFor('phone-transfer', Form),
     Form isReceived.
```

3.19.

Leavers should return the mobile device and any accessories including chargers to their line manager before their final working day. Failure to comply will result in the user being invoiced for the full cost of the modern equivalent handset and any other associated costs.

```
'nhs-trust' says 'line-manager' can-say
  Device hasBeenReturned.
```

3.20.

All smart devices issued by the Trust are done so on a contract basis. Managers / budget holders will be responsible for mobile device costs within their team. Therefore it is imperative that the IT Department is kept informed of any moves / changes as they occur to ensure this does not impact on the budget holder (including replacing devices in the event of loss).

```
'nhs-trust' says Manager isResponsibleFor(Device)
  if Device isOwnedBy(Staff),
     Manager isManagerOf(Staff).
```

3.21.

The IT Department will keep a register of devices and allocations and will monitor mobile device usage for excessive use and will bring any issues to the attention of the staff member and their manager.[3]

```
'nhs-trust' says 'it-department' can-say
  'nhs-trust' hasDevice(Device:D).
```

[3](See Requirement 4.4.4.)

3.22.

It is the responsibility of the individual to ensure that they adhere to signage and instructions governing the use of devices whilst within a public service property.

```
'nhs-trust' says Employee:S can-say
  S hasAcknowledged('public-usage-policy').
```

4. *Personal Usage of Corporately Issued Mobile Device*

4.1.

If a mobile device user wishes to make personal use of the device, this must be declared on the declaration of use form and a minimum of 5 per month should be paid by the user. This monthly charge is intended primarily to cover the cost of voice communication and texting for private use. The use of the mobile device for data usage over mobile networks for private use is discouraged. However,

```
'nhs-trust' says Staff canUseForPersonal(Device)
  if Device isOwnedBy(Staff),
     Device isOwnedBy('nhs-trust'),
     Device isPersonalUse,
     Staff hasAcknowledged('monthly-fee'),
     Staff hasAcknowledged('acceptable-use').
```

should a member of staff wish to access the web in their own time this would be acceptable.

4.2.

The Trust expects all members of staff who opt to use a corporately issued mobile device for personal usage to keep usage appropriate and legal at all times. It will be a disciplinary matter if the device were used inappropriately or in any illegal or unsavoury way in accordance with the principles of the NHS Constitution.

4.3.

The Trust would not encourage staff members to download apps for personal use onto a corporately issued mobile device or to use the device to store personal data.

4.4.

The IT Department will monitor the device usage for excessive use and will bring any issues to the attention of the staff member and their manager.

4.5. As stated earlier, in the event of loss of the device, all data including personal data, photographs and personal apps including paid apps will be remote wiped and the Trust would not recompense staff for loss of any such data or paid apps.

5. *Mobile Devices and Driving*

5.1.

For safety reasons, Trust staff must not use a hand held mobile device whilst driving any vehicle. It is illegal to do so. Please refer to the most up-to-date information via the Highways Agency.

5.2. It is not Trust policy to provide hands-free equipment and the Trust does not recommend using mobile devices in hands-free mode or with hands-free attachments whilst driving.

5.3.

The Trust will not take responsibility or be liable in any way for legal charges or other consequences of using a mobile device whilst driving.

6. *Roaming Arrangements and International barring*

6.1.

```
'nhs-trust' says 'it-department' canMonitor(Device)
  if Device isOwnedBy('nhs-trust').

'nhs-trust' says 'it-department' can-say
  Device isUsedExcessively.

'nhs-trust' says 'it-department' mustInform(Manager, 'excessive-use')
  if Device isOwnedBy(Staff),
     Device isUsedExcessively,
     Manager isManagerOf(Staff).
```

```
'nhs-trust' says Employee:S can-say
  S hasAcknowledged('data-loss-policy').
```

```
'nhs-trust' says Device:D mustDisable(D)
  where inCar(D) = true,
        usingHandsFree(D) = false.
```

```
'nhs-trust' says Employee:S can-say
  S hasAcknowledged('driving-policy').
```

```
'nhs-trust' says Device canCall(TelephoneNumber:X)
  if Device isOwnedBy('nhs-trust')
  where nationalNumber(X) = true,
        premiumNumber(X) = false.

'nhs-trust' says Device canCall(TelephoneNumber:X)
  if Device isOwnedBy(Staff),
     Staff canDuring(From, To, 'make-international-calls')
  where betweenDates(From, To) = true.

'nhs-trust' says 'it-department' can-say
  Staff canMakeInternationalCalls(From, To)
  if Staff hasAppliedFor('international-calls', Form),
     Manager hasApproved(Form).

'nhs-trust' says Manager can-say inf
Manager hasApproved('international-calls', App)
  if Staff hasAppliedFor('international-calls', App),
     Manager isManagerOf(Staff).
```

All mobile devices will be configured for national access only. Premium / international calls will be barred. International call barring and roaming arrangements can be lifted for specific periods, to be stipulated on request, on approval of the relevant manager / budget holder. This may be granted by emailing a member of the IT Department or via:

`tct.mobilephone@nhs.net`

giving the reason for the request. However, members of staff must be aware that if email is used whilst abroad it will cost extra money and the cost may be recoverable personally from the device holder. It is recommended that if going abroad, the device holder's phone is used for voice communication only. The IT and Telecommunications Team can assist with turning data off or they can arrange for a standard mobile phone to be loaned for the duration of the time abroad.

6.2. If a device holder is paying for personal use, use of data whilst abroad will not be sanctioned unless under exceptional circumstances to be agreed in advance with the line manager.

```
'nhs-trust' says Device mustDisable('international-calls')
  if Device isPersonalUse.
```

7. *Use of Camera Enabled Mobile Devices*

7.1.

Some mobile devices have the ability to take photographs / videos. This function should not be used for photographs / videos of an individual's care and treatment unless the device has encryption enabled and it is clinically appropriate to do so. Please refer to Appendix B; 'Use of Smart Devices for Photography and Videoing when Assessing and Planning Care'.

```
'nhs-trust' says Device canPhotograph(Patient)
  if Device isEncrypted,
     Patient isPhotographable.
```

7.2.

If the photography / video facility is used as part of the recording of an individual's care and treatment, the device user must ensure that the consent of the individual has been collected prior to taking any photograph / video. The individual needs to fully understand why the photograph / video is being taken and the member of staff plans to do with it, in particular if it will be shared. A record of the consent must be entered into the individual's care record. It would be good practice to show the individual the photograph / video once taken.

```
'nhs-trust' says 'clinician' can-say
  Patient isPhotographable
  if Patient hasConsentedTo('photography').

'nhs-trust' says Patient can-say inf
  Patient hasConsentedTo('photography')
  if Patient canConsent.

'nhs-trust' says 'clinician' can-say
  Patient canConsent.
```

8. *Smart Tablets*

Where appropriate the use of a Smart tablet may be considered a more appropriate device as opposed to a Smartphone. This will be determined on an individual basis at the discretion of the line manager and IT Department.

9. *Apps Management*

9.1.

Downloading of personal apps onto a corporately issued mobile device should be avoided where possible. The Trust would not encourage staff members to download apps for personal use onto a corporately issued mobile device. All staff are reminded that they must adhere to the guidance outlined in the Social Media Policy.

9.2.

Apps for work usage must not be downloaded onto corporately issued mobile devices (even if approved on the NHS apps store) unless they have been approved through the following Trust channels:

```
'nhs-trust' says App isInstallable
  if App hasMet('clinical-use-case').
'nhs-trust' says App isInstallable
  if App hasMet('business-use-case').
```

9.3.

Clinical apps; at the time of writing there are no apps clinically approved by the Trust for use with patients / clients. However, if a member of staff believes that there are clinical apps or other technologies that could benefit their patients / clients, this should be discussed with the clinical lead in the first instance and ratification should be sought via the Care and Clinical Policies Group. A clinical app should not be used if it has not been approved via this group.

```
'nhs-trust' says 'cacpg' can-say
  App hasMet('clinical-use-case').
```

9.4.

Business apps; at the time of writing there are no business (i.e., non-clinical) apps approved by the Trust for use other than those preloaded onto the device at the point of issue. However, if a member of staff believes that there are apps or other technologies that could benefit their non-clinical work, ratification of the app must be sought via the Management of Information Group (MIG). An app should not be used if it has not been approved via this group.

```
'nhs-trust' says 'mig' can-say
  App hasMet('business-use-case').
```

9.5.

Following approval through Care and Clinical Policies and / or MIG, final approval will be required through Integrated Governance Committee.

```
'nhs-trust' says App isInstallable
  if App isDownloadable,
     App hasMet('final-app-approval').

'nhs-trust' says 'igc' can-say
  App hasMet('final-app-approval').
```

9.6.

Use of paid apps must be agreed in advance with the device holder's line manager and there should be a demonstrable benefit.

```
'nhs-trust' says Device canInstall(App)
  if App isInstallable,
     App isApprovedFor(Device).

'nhs-trust' says Employee:Manager can-say
  App:A isApprovedFor(Device)
  if Manager isResponsibleFor(Device).
```

9.7. Whilst apps are a useful tool to aid in clinical decision making they should not be used as a sole basis for clinical decision making. It is the legal responsibility for the clinician to justify the treatment or procedure that they have undertaken. The sole use of an app to support this is not valid justification.

9.8.

Where an Apple device has been approved, an Apple ID is required to download apps, whether free or paid. Guidance on creating an Apple ID is here. The creation of an Apple ID should be independent from any personal accounts the employee may hold. The installation and use of iTunes is not required to download apps from the apps store. ITunes will not be supported on corporate devices.

10. *Policy Non-Compliance*

10.1.

Policy non-compliance will be regarded as serious or gross misconduct, which is likely to result in disciplinary action being taken.

11. *Policy Distribution and Application*

11.1.

To all managers and mobile device users.

11.2.

Managers are expected to apply this policy equally across all areas throughout the Trust.

11.3.

Associated forms and paperwork will be maintained and kept up-to-date. It should be noted that the forms contained within the appendices may be updated from time to time.

```
'nhs-trust' says 'clinician' can-say
  Treatment:T isJustifiedBy(String:Reason).
```

A.2 SANS

2 Security Policy for Handheld Devices

2.1 General policy requirements

Policy agreement.

IT department MUST ensure that all employees (regular employees, interns, externals) using devices falling into the category “handheld devices” as defined in section 1.7, have acknowledged this security policy and the associated procedures before they are allowed to use corporate services using handheld devices.

```
'company' says 'it−department' can-say∞ Employee:U canUse(Device:D).

'it−department' says Employee:User canUse(Handheld:Device)
 if U hasAcknowledged('policy').

'it−department' says Employee:User can-say
 User hasAcknowledged('policy').

'it−department' says Employee:U mustAcknowledged('policy').

'it−department' says 'regular' isEmployee.
'it−department' says 'intern' isEmployee.
'it−department' says 'external' isEmployee.

'it−department' says 'pocket−pc' isHandheld.
'it−department' says 'smartphone' isHandheld.
```

Use of private handheld in corporate environment. IT governance MUST define whether private handhelds are authorized to connect to corporate networks in the user acceptance policy, according to its risk assesment policy.

```
'company' says 'it−governance' can-say Device canConnectToNetwork('corporate−network')
 if Device hasMet('risk−assesment−policy').
```

Private handhelds are not authorized: In highly restricted facilities, private handheld devices MUST be prohibited. In that case, mobile devices MUST be collected prior to the user’s entrance into the facility.

Private handhelds are authorized in offices but are not allowed to connect to internal networks.

Private handhelds MUST NOT connect to corporate networks and access corporate information. This includes synchronization with a workstation connected to internal networks. Corporate networks MUST be protected accordingly using network access control mechanisms and MUST NOT grant access to any corporate information to unregistered devices.

```
'company' says Device mustProhibitCollectAt(Location)
 if Device isPrivatelyOwned,
    Location isSecurityLevel('restricted').
```

Private handhelds are authorized: Any non business-owned (that, is private) device must be able to connect to ⟨Company⟩ network MUST first be approved by technical personnel such as those from the ⟨Company⟩ IT department or desktop support.

If allowed, privately-owned handheld devices MUST comply with this security policy and MUST be inventoried along with corporate handhelds, but identified as private. This is in order to prevent theft of corporate data with unmanaged handhelds (i.e. owner of device is not identified).

```
'company' says Device canConnectToNetwork('company−network')
 if Device isApproved,
    Device isActivated,
    Device isPrivatelyOwned.

'company' says 'technical−personnel' can-say Device isApproved.

'company' says 'it−department' can-act-as 'technical−personnel'.
'company' says 'desktop−support' can-act-as 'technical−personnel'.
```

IT department roles and responsibilities. IT governance is responsible for the mobile handheld device policy at ⟨Company⟩ and shall conduct a risk

```
'company' says 'it−department' can-say Device:X hasMet('mobile−handheld−device−policy').
'company' says 'it−department' can-say Device:X hasMet('approved−device−policy').
'company' says 'it−department' can-say App:A isInstallable.
'company' says 'it−department' can-say App:A isNotInstallable.
```

analysis to document safeguards for each device type to be used on the network or on equipment owned by ⟨Company⟩.

The IT department maintains a list of approved mobile handheld devices and makes the list available on the intranet. The IT Department maintains lists of allowed and unauthorized applications and makes them available to users on the intranet.

2.2 Physical security

Physical security. In case of loss or theft of handheld, users MUST report AS SOON AS POSSIBLE (right after the loss has been noticed) the IT department or help desk, in order to take the appropriate measures.

Procedure for reporting lost device MUST exist and be clearly communicated to all users:

To report lost or stolen mobile computing and storage devices, call the Enterprise Help Desk at +41-xx-xxx-xx- xx. For further procedures on lost or stolen handheld wireless devices, please see the PDA Information and Procedures section.

```
'company' says User can-say
  Device isLost
  if Device isOwnedBy(User).

'company' says User mustInform('enterprise-help-desk', 'device-lost')
  if D isOwnedBy(User),
     D isLost.
```

Device safety. Usage of handheld devices in uncommon situations is depicted in the acceptable use policy 0, which states that: Conducting telephone calls or utilizing handhelds while driving can be a safety hazard. Drivers should use handhelds in hand only while parked or out of the vehicle.

If employees must use a handheld device while driving, ⟨Company⟩ requires the use of hands-free headset devices.

```
'company' says Device:D canCall(TelephoneNumber:X)
  where inCar(D) = true,
        usingHandsfree(D) = true.
```

This seems very similar to section 5 of the NHS device policy.

Password policy. Access to handheld devices MUST be password-protected.

```
'company' says Device:D hasMet('password-policy')
  where passwordEnabled(D) = true.
```

Ownership information. Owner information SHALL be written on the handheld. Owner should be either end-user (if users are responsible for their device) or generic owner information to avoid revealing the company name and thus exposing the device to more scrutiny. This is possible in two ways, according to hardware capabilities:

```
'company' says Device can-say Device isOwnedBy(X)
  if Device isOwnedBy(X).

'company' says Device:D can-say D isOwnedBy(Company).
```

- Either the information can be displayed on the lockout screen on the handheld

- Or the information MUST be written on a sticker on the back of the handheld

This would allow anyone finding a lost device to return it to its owner.

Availability of device & services—business continuity. As handheld devices consume lots of resources (processing, memory), battery management is crucial to ensure business continuity. Mobile users working out of companys offices MUST have the necessary accessories to charge their device, according to the situation they are in: car, train, at customer sites, etc.

Batteries are consumed faster during the following operations, and devices should be switched off if not used:

- Wireless LAN (searching for nearby network)
- Encryption/decryption of communications

```
'company' says Device:D mustDisable('wlan')
  where usingBattery() = true.

'company' says Device:D mustDisable('encryption')
  where usingBattery() = true.
```

Similar to 3.11–3.13 of NHS policy.

Use of camera. Digital camera embedded on handheld devices might be disabled in restricted environments, according to ⟨Company⟩ risk analysis. In sensitive facilities, information can be stolen using pictures and possibly sent using MMS or E-mail services.

In high-security facilities such as R&D labs or design manufacturers, camera MUST be disabled. Furthermore, MMS messages should be disabled as well, to prevent malicious users from sending proprietary pictures.

```
'company' says Device mustDisable('camera')
  if ' restricted −environment' isWhereIs(Device).

'company' says Device mustDisable('mms')
  if 'high−security−environment' isWhereIs(Device).

'company' says 'high−security−environment' can-act-as 'restricted−environment'.

'company' says Location:L isWhereIs(Device:D)
  where location(D, L) = true.
```

2.3 Operating system security

Firmware version, updates & patching. Devices firmware MUST be up-to-date in order to prevent vulnerabilities and make the device more stable. Firmware patching and updating processes are the responsibility of the IT department, MUST be documented and tested prior to deployment on a whole fleet of handsets

```
'company' says Device mustBeUpdated
  if Device isRunningVersion(Version)
  where lth(deviceVersion(Device), Version) = true.

'company' says 'it−department' can-say Device:D mustRun(Version:V).
```

OS hardening: removing unnecessary services. In order to enhance the security level of end devices, all unnecessary built-in services should be disabled, especially including:

- Internet file-sharing
- FTP client

```
'company' says 'device' mustDisable('file−sharing').
'company' says 'device' mustDisable('ftp−client').
```

How do we check for this? Should device query if ∃ service that should be disabled?

System hardening: removing unnecessary applications. If employees have no reason to use certain file types (especially MP3s and videos), removal of the corresponding applications from the devices is recommended.

This not only prevents a devices being used as an expensive MP3 player, but it also protects the organization from potential legal problems regarding these types of media (DRMs infringement). Furthermore, removing unnecessary applications prevents attackers from exploiting implementation flaws in those applications.

```
'company' says App isNotInstallable
  if App isAssociatedWith('mp3').

'company' says App isNotInstallable
  if App isAssociatedWith('mimetype—audio—mpeg').
```

I need to fix bug in parsing due to unacceptable characters in strings (i.e. '*' and '.')

Unsigned applications policy. Users MUST NOT install any UNSIGNED application or applications theme on the handheld device, for any purpose; this in policy order to prevent malicious infection of the device.

```
'company' says App hasMet('unsigned—applications—policy')
  if App isSignedBy(X).
```

Certificates management. Only IT department staff are authorized to manage (install and revoke) certificates on handhelds. The IT department MUST provide the necessary certificates to enable all required services to users. Only the IT department can install certificates in the root certificates store or in the intermediate certificates store (if available).

```
'company' says 'it—department' can-say Certificate:C isValid.
```

Antivirus policy. Mobile devices MUST have antivirus software installed to prevent viruses from being vectored into the corporationeither as e-mail attachments or through file transfers. Antivirus software MUST be configured in order to:

- Do automatic signature update when connected to desktop PC or wireless network
- Do automatic and regular scan of device

```
'company' says 'av—software' canConnectToServer(URL:X)
  where connectedToWifi() = true.

'company' says 'av—software' canScan(Device:D).
```

There is a delegation relationship to the AV software in terms of its fuctionality and the requirement to have it installed, but I'm unsure how to express it. Software traditionally doesn't speak in AppPAL.

2.4 Personal Area Networks (PAN) security policy

Bluetooth version. No Bluetooth Device shall be deployed on ⟨Company⟩ equipment that does not meet Bluetooth v2.1 specifications without written authorization from the Information Security Manager.

Any Bluetooth equipment purchased prior to this policy MUST comply with all parts of this policy except the Bluetooth version specifications.

```
'company' says Device:D canEnable('bluetooth')
  where geq(bluetoothVersion(), '2—1') = true.

'company' says 'information—security—manager' can-say Device canEnable('bluetooth').
```

```
'company' says Device1 can-say Device1 hasPairedWith(Device2, PIN:P)
  if Device1 canPair(Device2).

'company' says Device:D1 canPair(Device:D2)
  where location(D1, 'restricted') = false.
```

The second part (re: reporting duplicate pairing attempts) seems tricky to express in AppPAL. Feels like a meta-policy about the state of a device's assertion context.

PAN PINs and pairing. When pairing two communicating devices in a PAN, users should ensure that they are not in a public area. If the equipment asks for a PIN after it has been initially paired, users MUST refuse the pairing request and immediately report it to IT department or the help desk. Unless the device itself has malfunctioned and lost its PIN, this is a sign of a hack attempt.

Care must be taken to avoid being recorded when pairing Bluetooth adapters; Bluetooth 2.0 Class 1 devices have a range of 100 meters.

File transfer (beam in PAN). File transfers between devices in close range (PAN), taking place over Bluetooth or Infrared, MUST take place only between authenticated parties, which MUST agree on a pairing key as defined in section 4.2: PAN PINs and pairing.

Anonymous connections (i.e. without pairing) MUST NEVER take place.

```
'company' says Device1 canSendTo(Device2, Data:X)
  if Device1 hasPairedWith(Device2, PIN),
     Device2 hasPairedWith(Device1, PIN).

'company' says Device1 can-say Device1 hasPairedWith(Device2, PIN)
  if Device2 hasStartedPairing(Device1, PIN).
```

PAN security audits. Information security staff SHALL perform audits for Bluetooth and IrDA to ensure compliance with this policy. In the process of performing such audits, information security auditors SHALL NOT eavesdrop on any phone conversation.

```
'company' says 'is-staff' canMonitor(Device:D, Feature:X)
  where ! X = 'conversation'.
```

Similar to NHS policy 3.14.

Infrared IrDA. Infrared support MUST be disabled if Bluetooth connectivity is supported. Bluetooth MUST be preferred to IrDA when available.

```
'company' says Device mustDisable('irda')
  if 'bluetooth' isOwnedBy(Device).
```

2.5 Data security

Information classification. The information classification policy applies restrictively to handheld devices as it applies to laptops.

A handheld device SHALL NOT be used to enter or store passwords, safe/-door combinations, personal identification numbers, or classified, sensitive, or proprietary information. Corporate documents are classified according to their level of confidentiality e.g.:

- Public documents
- Internal documents
- Confidential documents
- Strictly confidential or secret documents

```
'company' says 'device' cannotStore(Doc) if Doc isSecurityLevel('secret').
'company' says 'device' cannotStore(Doc) if Doc isSecurityLevel('strict-confidential').
'company' says 'device' cannotStore(Doc) if Doc isSecurityLevel('confidential').
'company' says 'device' cannotStore(Doc) if Doc isSecurityLevel('internal').
```

- Secret and confidential documents MUST NEVER be stored on end devices.

Internal documents SHOULD NOT be stored on mobile devices unless strictly necessary. Public documents can be carried on mobile devices without risk. However, if not needed, public corporate files MUST be removed from the device.

Data security. Mobile handheld devices containing confidential, personal, sensitive, and generally all information belonging to ⟨Company⟩ SHALL employ encryption or equally strong measures to protect the corporate data stored on the device, as stated in corporate encryption standards.

If memory encryption is not available natively in the device, a third party application SHALL be purchased.

```
'company' says Encrypted:Device canStore(Confidential:Doc).
```

Persistent memory. Corporate data, i.e. any corporate file, even public, MUST not be stored in persistent (or device) memory, but rather in memory card (SD or MMC).

```
'company' says 'internal' canStore(File:Doc).
'company' says 'external' cannotStore(File:Doc).
```

We could do this with app permissions, by prohibiting the `WRITE_EXTERNAL` permission.

Encryption of removable storage card. Removable storage on smartphones (e.g. SD cards) MUST be encrypted in order to prevent data theft on storage card.

Usually, encryption of MMC is natively built in devices. Encryption of MMC MUST be turned on. Third-party encryption software might be used if native platform does not offer the option of data encryption on MMC.

```
'company' says Device:D canInstall(App)
  if App isEncrypting,
     App isInstallable.

'company' says Device canInstall(App)
  if App isInstallable,
     Device mustEnable('mmc-encryption').

'company' says Device:D can-say
  D mustEnable('mmc-encryption').
```

2.6 Corporate networks access security

Network access control. All devices, including handhelds that have to connect to internal networks MUST be identified by IT department for Network Access Control purposes, after they are declared in the corporate inventory.

Any attempt to access corporate networks with an unknown device will be considered as an attack against corporate assets.

```
'company' says 'it-department' can-say Device canConnectToNetwork('internal')
  if Device isActivated.

'company' says 'it-department' can-say Device:D isActivated.
```

File sharing. File-sharing services MUST be disabled, independently of the transport technology.

```
'company' says Device:D mustDisable('file-sharing').
'company' says Device:D canEnable('file-sharing')
  if 'file-sharing' mustEnable('authentication').
```

If enabled, authentication MUST be in place to force the identification of the communicating party: no anonymous access shall be possible. Guidelines or a policy depicting valid passwords are available in the password policy.

Wireless support. Independently of the company risk analysis, disable WLAN support in the following cases: Whenever connectivity is not required to prevent unnecessary battery consumption.

When connected to a desktop computer to prevent the spread of malware.

Access to WLAN MUST be restricted if mobile workers do not require access to public, open, or untrusted WLAN, according to ⟨Company⟩ risk analysis and its business model:

- Restrict the list of authorized access points to corporate access points only.
- Disable connection to open/public WLANs without encryption and authentication methods.
- Disable connecting to WEP-protected WLANs (considered insecure).

```
'company' says 'device' mustDisable('wifi').

'company' says Device:D canConnectToAP(AP:X)
  if X isOwnedBy('company').

'company' says Device:D canConnectToAP(AP:X)
  if X canAuthenticateWith('wpa').
```

2.7 Over-the-air provisioning security

Handheld configuration for OTA provisioning. Mobile devices MUST be configured to receive provisioning files only from a list of trusted PPGs. This white list of trusted PPGs MUST contain corporate PPG IP address only, and eventually other PPGs owned by the operator.

This white list MUST be provisioned by security staff, and regularly pushed to end devices for security policy enforcement.

```
'company' says PPG:PPG can-say File:F isProvisioningFile
  if PPG isApproved.
'company' says 'security-staff' can-say PPG:PPG isApproved.
```

OTA provisioning messages security. Provisioning messages (using OMA DM or WAP Push) MUST be encrypted. Necessary SSL certificates MUST be provided by corporate IT department. Note that SSL certificates on OMA Device Management Server must be signed by a Certification Authority or by the Operator.

```
'company' says 'it-department' can-say Device:D isUpdatedBy(Update:U).
'company' says 'operator' can-say Server:S isUpdatedBy(Update:U).
```

2.8 Internet Security

Use of Internet services. Users MUST agree to the email security/acceptable use policy and eventually to the eCommerce security policy.

```
'company' says User canUse(Device)
  if Device isOwnedBy(User),
     User hasAcknowledged('email-security'),
     User hasAcknowledged('acceptable-use'),
     User hasAcknowledged('ecommerce-security').

'company' says Employee:U mustAcknowledged('email-security').
'company' says Employee:U mustAcknowledged('acceptable-use').
'company' says Employee:U mustAcknowledged('ecommerce-security').
```

Seems to add to the `canUse` policy in 2.1.

E-mail attachments download. Users SHALL NOT download files attached to e-mails. Restriction of attachment downloading can be implemented on both the device (via provisioning) and the mobile email server (via configuration). Attachment download restrictions MUST be implemented, preferably in mobile e-mail servers in order to prevent users tweaking the device security features.

```
'company' says Device:D mustDisable('attachment−download').
```

A.3 HiMSS

Mobile Security Toolkit: Sample Mobile Device User Agreement

Healthcare Information and Management Systems Society

February 27, 2017

Introductory Note: *The sample mobile device user agreement is an example of an agreement that is being used by a health system to manage personal mobile devices in its environment. It is only an example and is not meant to be a complete or exhaustive list of policy elements. Because organizations, along with regulatory and legal requirements, are different, each organization should develop a unique mobile device user agreement that is aligned with the needs of the organization, applicable laws, and is consistent with its policies and procedures.*

As a condition of synchronizing my personal PDA/Smartphone with the XYZ Health System computing environment, I understand that I am subject to certain restrictions and expectations on the use of my PDA/Smartphone. This document serves as notification of the restrictions and expectations, and acceptance and acknowledgement thereof.

By signing below:

- I agree to follow all XYZ Health System policies relating to the use and security of portable computing devices.
- I acknowledge that XYZ Health System will enforce security settings on the PDA/Smartphone including at a minimum encryption of all XYZ Health System information on the device.
- I understand that if I do not make appropriate backups of my *personal information* maintained on the PDA/Smartphone in order to avoid loss of information should the device be lost, stolen, corrupted, or data must be


```
'user' says 'xyz-health-system' can-say
  Device:D mustSet(SecuritySetting:Opt, Value)
  if Value isValueFor(Opt).
```

```
'xyz-health-system' says 'user' mustAcknowledged('data-loss-policy').
'user' says 'user' hasAcknowledged('data-loss-policy').
'user' says 'xyz-health-system' can-say
  Device mustWipe
  if Device isOwnedBy('user').
```

deleted (wiped) in order to protect sensitive information, such personal information may be lost and is not the responsibility of XYZ Health System.

- I agree not to backup XYZ Health System information (including e-mail) to a non-XYZ Health System computer or move the XYZ Health System information from its encrypted area to any other areas on the smart phone. I understand my XYZ Health System email mailbox information is maintained and backed up by XYZ Health System and should not be replicated onto non-XYZ Health System computers.

```
'user' says Device:D canBackupTo(Computer)
  if Computer isOwnedBy('xyz—health—system').
'user' says 'xyz—health—system' can-say
  'xyz—health—system' hasDevice(Device:D).
```

- I agree to hold XYZ Health System harmless for any loss relating to the administration of PDA/Smartphone connectivity to XYZ Health System systems including, but not limited to, loss of personal information stored on a PDA/Smartphone due to data deletion done to protect sensitive information related to XYZ Health System, its patients, members or partners.

```
'xyz—health—system' says 'user' mustAcknowledged('data—loss—policy').
'user' says 'user' hasAcknowledged('data—loss—policy').
```

- I understand that modifying the underlying operating system of the device (e.g., "rooting", "Jailbreak-ing", etc.) will result in the device being removed from synchronization with XYZ Health System data and voids this agreement and support for the device.

```
'xyz—health—system' says 'user' mustAcknowledged('rooting—policy').
'user' says 'user' hasAcknowledged('rooting—policy').
```

- I understand and accept that synchronization relies on one or more cellular network providers and the Internet, and that both are subject to slowdowns and outages of extended duration that are beyond the control of IT. Service cannot be guaranteed or fixed by XYZ Health System.

```
'xyz—health—system' says 'user' mustAcknowledged('internet—unreliable').
'user' says 'user' hasAcknowledged('internet—unreliable').
```

- I understand that access to the XYZ Health System wireless network is not available to non-XYZ Health System devices. All connectivity must be through cellular provider.

```
'user' says Device canConnectToAP(AP)
  if Device isOwnedBy('user'),
     Device isOwnedBy('xyz—health'),
     AP isOwnedBy('xyz—health—system').
```

- I agree not to transmit XYZ Health System sensitive information (e.g., Business Sensitive Information (BSI) or Protected Health Information (PHI)) through non-XYZ Health System approved methods. These include texting, paging, personal email and social networks. Electronic communications with patients should be through MyXYZHealthSystem. BSI can be transmitted using secure e-mail (e-secure) of secure file transfer methods.

```
'user' says Data canBeSentWith('MyXYZHealthSystem')
  if Data isSecurityLevel('protected—health').

'user' says Data canBeSentWith('e—secure')
  if Data isSecurityLevel('business—sensitive').

'user' says Data canBeSentWith('secure—ftp')
  if Data isSecurityLevel('business—sensitive').
```

```
'xyz—health—system' says 'user' mustAcknowledged('diagnostic—limitations').
'user' says 'user' hasAcknowledged('diagnostic—limitations').
'user' says App canStore(Data)
  if Data isSecurityLevel('protected—health'),
     App isApprovedFor(Device).

'user' says 'xyz—health—system' can-say
  App:A isApproved.
```

- I understand that these devices should not be considered diagnostic quality for patient care decisions, and should not contain Protect Health Information (PHI), unless incorporated as part of an officially approved, standard application support by XYZ Health System.

- I agree that the PDA/Smartphone can be wiped by XYZ Health System upon the decision of XYZ Health System management and understand that it will delete all data including personal files.

- I understand that there is a one-time charge to my department for activation and management of a new device, which includes replacements and upgrades: Blackberry $xxx, Android/iPhone $xxx.

- I agree to report loss of a device immediately to the IT Help Desk xxx-xxx-xxxx.

- I understand that non-exempt employees carrying or operating a PDA/Smartphones device outside of normal work hours does not constitute working remotely unless properly authorized by management.

- I understand that failure to adhere to these conditions or failure to appropriately safeguard XYZ Health System information could result in action against me personally, including termination of employment, civil action (e.g., being sued directly) or criminal prosecution by effected persons. [This exposure is especially relevant to disclosure of Protected Health Information ("PHI") or Business Sensitive Information ("BSI")].

```
'user' says 'xyz-health-system' can-say
  Device mustWipe
  if Device isOwnedBy('user').
```

```
'xyz-health-system' says 'user' mustAcknowledged('activation-charge').
'user' says 'user' hasAcknowledged('activation-charge').
'department' says Device isActivated
  if Device isOwnedBy(User),
     User hasDepartment('department'),
     Device hasActivationFee(Fee),
     User hasPaid(Fee).
```

```
'user' says 'user' mustInform('it-help-desk', 'device-lost')
  if Device isOwnedBy('user'),
     Device isLost.
```

```
'xyz-health-system' says 'user' mustAcknowledged('remote-working-policy').
'user' says 'user' hasAcknowledged('remote-working-policy').
```

```
'xyz-health-system' says 'user' mustAcknowledged('penalty-conditions').
'user' says 'user' hasAcknowledged('penalty-conditions').
```

A.4 Edinburgh

BYOD Policy: Use of Personally Owned Devices for University Work

University of Edinburgh Records Management

March 20, 2017

1 Audience and purpose

1.1

This policy is for all staff using personally owned devices such as smart phones, tablet computers, laptops, netbooks and similar equipment, to store, access, carry, transmit, receive or use University information or data, whether on an occasional or regular basis. The term for such devices is BYOD ("bring your own device").

1.2

The University recognises the benefits brought by the use of your own devices in work and welcomes it. This policy is about reducing the risk in using BYOD. Such risks may come from your BYOD being lost, stolen, used or exploited in such a way to take advantage of you or the University.

1.3

This policy sets out the minimum requirements. Individual business areas may specify additional, higher requirements as necessary.

```
'records-management' says BA can-say
  Device hasMet('higher-requirements')
  if Device isOwnedBy(Employee),
     Employee hasDepartment(BA).
```

1.4

We believe that following the procedures set out below will bring benefits to staff through protection of your own data as well as that of the University.

2 General principle

2.1

If you use your own device for University work, it is important to ensure that it and the information it contains is appropriately protected.

3 Data sensitivity

3.1

The Universitys Policy on Taking Sensitive Information and Personal Data Outside the Secure Computing Environment provides guidance on categories of high and medium risk personal data and business information.

3.2

If some of your work involves the use of high or medium risk information, and you use a BYOD, it is likely that some of it will find a way on to your device, for example within your email, or if you are working on documents away from your office.

4 Requirements for users of high and medium risk information; Advice for low risk users

4.1

High and medium risk users: You are required to comply with all bullet points listed below to permit use of BYOD for work. If necessary, seek help from your IT support personnel to meet these requirements.

```
'records-management' says Device hasMet('information-policy')
  if Employee has(Device),
    Employee hasMet('high-risk'),
    Device hasMet('high-risk-any-device-policy'),
    Device hasMet('high-risk-specific-device-policy').

'records-management' says Device hasMet('information-policy')
  if Employee has(Device),
    Employee hasMet('low-risk'),
    Device hasMet('low-risk-any-device-policy'),
    Device hasMet('low-risk-specific-device-policy').
```

4.2

Low risk users: For protection of your own data as well as low risk work data, you are advised to comply with at least the triangle bullet points below. Consider what the potential consequences could be for you, your friends or your family should your device become lost or stolen, and what protection configuration you want to put in place to prevent your data from being misused.

4.3

If you are in any doubt about which of the above classes you fall into you must assume that you are a High and medium risk user. Experience shows that almost all academic staff and those in HR, Finance and student-related roles will be high and medium risk users.

Any type of device

- △ Set and use a passcode (e.g. pin number or password) to access your device. Whenever possible, use a strong passcode. Do not share the passcode with anyone.
- △ Set your device to lock automatically when the device is inactive for more than a few minutes.
- △ Take appropriate physical security measures. Do not leave your device unattended.
- △ Keep your software up to date.
- △ Make arrangements to back up your documents.
- △ Keep master copies of work documents on a University managed storage service.
- If other members of your household use your device, ensure they cannot access University information, for example, with an additional account passcode. (Our preference is for you not to share the device with others.)
- Organise and regularly review the information on your device. Delete copies from your device when no longer needed.

```
'records-management' says Device hasMet('low-risk-any-device-policy')
  if Device hasPassword(Password),
     Password isStrongPassword,
     Device mustLockAfter(N),
     Device canBackupTo(Server),
     Server isOwnedBy('university')
  where leq(N, '180') = true.

'records-management' says Device hasMet('high-risk-any-device-policy')
  if Device hasMet('low-risk-any-device-policy'),
     Device isOwnedBy(Employee),
     Employee mustAcknowledged('erase-on-loss'),
     Employee hasAcknowledged('erase-on-loss'),
     Device isEncrypted.
```

- When you stop using your device (for example because you have replaced it) and when you leave the Universitys employment, securely delete all (non-published) University information from your device.
- Encrypt the device (to prevent access even if someone extracts the storage chips or disks and houses them in another device)[1][2].
- Report any data breaches in accordance with the Incident Reporting Policy.
- Configure your device to maximise its security. For example each new technology brings new enhanced security features. Take time to study and discover how to use these and decide which of them are relevant to you. Seek help from your IT support team if necessary.
- Whenever possible, use remote access facilities to access information on University systems. Log out and disconnect at the end of each session.

[1] If your device is an Apple iPhone or iPad, it is encrypted and protection is effective as soon as you set a PIN locking code.

[2] If your device is Android, there is an option to turn on whole-device encryption in its configuration settings. Other devices may or may not be encryptable. We recommend that you include your ability to encrypt as a factor when you are choosing your own devices.

Mobile phones, smart phones and "tablet" devices

△ Configure your device to enable you to remote-wipe it should it become lost[3].

△ If your device is second hand, restore to factory settings before using it for the first time.

- Only download applications ('apps') or other software from reputable sources.

```
'records−management' says SmartPhone:Device hasMet('low−risk−specific−device−policy')
  if Device hasFeature('remote−wipe').

'records−management' says Device hasMet('high−risk−specific−device−policy')
  if Device hasMet('low−risk−specific−device−policy'),
    Device cannotSideload.
```

[3] If you configure your device for use with Office365, you are able to remote wipe it using a service within the university.

Laptops, computers and more sophisticated tablet devices

△ Use anti-virus software and keep it up to date

```
'records−management' says Computer:Device hasMet('low−risk−specific−device−policy')
  if Device has(Antivirus:AV)
  where update(AV) = true.
```

Using wireless networks outside the University

△ Control your devices connections by disabling automatic connection to open, unsecured Wi-Fi networks and make risk-conscious decisions before connecting.

△ Disable services such as Bluetooth and wireless if you are not using them.

```
'records−management' says Device:D mustDisable('automatic−connection').
```

5 Consequences of non-compliance

5.1

The loss, theft or misuse of a BYOD is personally distressing. If you use sensitive data, it can also have serious consequences for others, for example staff and students about whom information is held. In addition there may be significant legal, financial and reputational consequences for the University, including fines of up to £500,000 can be levied. *You may also carry personal responsibility which, in serious cases could result in disciplinary action under the University Computing Regulations.*

```
'records-management' says Employee:U can-say
  U hasAcknowledged('compliance-policy').

'records-management' says Employee:U mustAcknowledged('compliance-policy').
```

A.5 Sirens

Sample BYOD Policy

http://www.code3pse.com/public/media/22845.pdf

February 27, 2017

1 Expectation of Privacy

⟨Department Name⟩ will respect the privacy of your personal device and will only request access to the device by technicians to implement security controls or to respond to legitimate discovery requests arising out of administrative, civil, or criminal proceedings. This differs from policy for ⟨Department Name⟩ provided equipment and/or services, where employees do not have the right, nor should they have the expectation, of privacy while using equipment and/or services.

```
'department' says 'technician' canMonitor(Device)
  if Device mustBeProvisioned.

'department' says 'technician' canMonitor(Device)
  if Device mustBeInvestigated.
```

2 Acceptable Use

2.1

The company defines acceptable business use as activities that directly or indirectly support the business of ⟨Department Name⟩.

2.2

The company defines acceptable personal use on company time as reasonable and limited personal communication or recreation, such as reading or game playing.

2.3

Devices may not be used at any time to:

```
'department' says Employee:E can-say
  E hasAcknowledged('acceptable-usage').

'department' says Employee:E mustAcknowledged('acceptable-usage').
```

- Store or transmit illicit materials
- Store or transmit proprietary information
- Harass others
- Engage in outside business activities
- Etc.

2.4

Employees may use their mobile device to access the following company-owned resources:

- Email
- Calendars
- Contacts
- Documents
- Etc.

```
'department' says Device:D canAccess('email').
'department' says Device:D canAccess('calendars').
'department' says Device:D canAccess('contacts').
'department' says Device:D canAccess('documents').
```

2.5

⟨Department Name⟩ has a zero-tolerance policy for texting or emailing while driving and only hands-free talking while driving is permitted.

```
'department' says Device mustEnable('hands—free')
  if Device isInCar.

'department' says Device mustDisable('speaker')
  if Device isInCar.

'department' says Device mustDisable('sms')
  if Device isInCar.

'department' says Device mustDisable('email')
  if Device isInCar.
```

3 Devices and Support

3.1

The following devices are supported:

- iPhone (3GS, 4, 4S, 5, etc...)
- iPad (⟨list acceptable models⟩)
- Android (⟨list acceptable models⟩)

```
'department' says IPhone3GS:Device isSupported.
'department' says IPhone4:Device isSupported.
```

- Blackberry (⟨list acceptable models⟩)
- Windows (⟨list acceptable models⟩)
- Etc...

3.2

Connectivity issues are supported by IT; employees should contact the device manufacturer or their carrier for operating system or hardware-related issues.

```
'department' says Employee:E can-say
  E hasAcknowledged('internet−unreliable').

'department' says Employee:E mustAcknowledged('internet−unreliable').
```

3.3

Devices must be presented to IT for proper job provisioning and configuration of standard apps, such as browsers, office productivity software and security tools, before they can access the network.

```
'department' says Device mustInstall(App)
  if Device mustBeProvisioned,
     App isStandardInstall.

'department' says 'it−department' can-say
  App:A isStandardInstall.
```

4 Security

4.1

In order to prevent unauthorized access, devices must be password protected using the features of the device and a strong password is required to access the company network.

```
'department' says Device:D mustEnable('password').
'department' says Device canConnectToNetwork
  if Device mustEnable('password').
```

4.2

The companys strong password policy is: Passwords must be at least six characters and a combination of upper- and lower-case letters, numbers and symbols. Passwords will be rotated every 90 days and the new password cant be one of 15 previous passwords.

```
'department' says String:Password isAcceptable
  if Password isCreated(Date)
  where geq(length(Password), '6') = true,
        largeAlphabet(Password) = true,
        leq(minus(today(), Date), '90') = true,
        novel(Password) = true.

'department' says Employee:U can-say
  Password:P isCreated(Date:D)
    where geq(D, delegatedAssertionDate()) = true.
```

The `delegatedAssertionDate` would probably have to be an obligation constraint, and would require some dark magic to implement.

4.3

The device must lock itself with a password or PIN if its idle for five minutes.

```
'department' says Device:D mustLock
  where geq(idleTime(), '300') = true.

'department' says Device cannotConnectToNetwork
  if Device isRooted.
```

4.4

Rooted (Android) or jailbroken (iOS) devices are strictly forbidden from accessing the network.

4.5

Smartphones and tablets that are not on the companys list of supported devices are not allowed to connect to the network.

```
'department' says Device cannotConnectToNetwork
  if Device isNotSupported.
```

4.6

Smartphones and tablets belonging to employees that are for personal use only are not allowed to connect to the network.

```
'department' says Device cannotConnectToNetwork
  if Device isNotProvisioned.
```

4.7

Employees access to company data is limited based on user profiles defined by IT and automatically enforced.

```
'department' says Device canReadData(X)
  if Device isOwnedBy(User),
     User canReadData(X).

'department' says 'it-department' can-say
  Employee:U canReadData(Data:D).
```

4.8

The employee's device may be remotely wiped if:

- The device is lost or stolen.
- The employee terminates his or her employment.
- IT detects a data or policy breach, a virus or similar threat to the security of the companys data and technology infrastructure.

```
'department' says Device mustWipe
  if Device isLost.

'department' says Device mustWipe
  if Device isOwnedBy(User),
     User isNotEmployed.

'department' says 'it-department' can-say
  Device:D mustWipe.
```

5 Risks/Liabilities/Disclaimers

5.1

While IT will take every precaution to prevent the employees personal data from being lost in the event it must remote wipe a device, but it is the employees responsibility to take additional precautions, such as backing up email, contacts, etc.

```
'department' says 'it-department' can-say
  Device:D mustWipe.
```

5.2

The company reserves the right to disconnect devices or disable services without notification.

```
'department' says 'company' can-say Device:D isDisconnected.
'department' says 'company' can-say Device:D mustDisable(Feature:X).
```

5.3

Lost or stolen devices must be reported to the company within 24 hours. Employees are responsible for notifying their mobile carrier immediately upon loss of a device.

```
'department' says Employee mustInform('company', 'device-lost')
  if Device isOwnedBy(Employee),
     Device isLost.
'department' says Employee mustInform(Carrier, 'device-lost')
  if Device isOwnedBy(Employee),
     Device hasCarrier(Carrier),
     Device isLost.

'department' says Employee can-say
  Device isLost
  if Device isOwnedBy(Employee).
```

5.4

The employee is expected to use his or her devices in an ethical manner at all times and adhere to the companys acceptable use policy as outlined above.

```
'department' says Employee:E can-say
  E hasAcknowledged('ethical-policy').

'department' says Employee:E mustAcknowledged('ethical-policy').
```

5.5

The employee is personally liable for all costs associated with his or her device.

```
'department' says Employee:E can-say
  E hasAcknowledged('financial- liabiliy -policy').

'department' says Employee:E mustAcknowledged('financial-liabiliy-policy').
```

5.6

The employee assumes full liability for risks including, but not limited to, the partial or complete loss of company and personal data due to an operating system crash, errors, bugs, viruses, malware, and/or other software or hardware failures, or programming errors that render the device unusable.

```
'department' says Employee:E can-say
  E hasAcknowledged('liabiliy-policy')

'department' says Employee:E mustAcknowledged('liabiliy-policy')
```

5.7

⟨Department Name⟩ reserves the right to take appropriate disciplinary action up to and including termination for noncompliance with this policy.

6 User Acknowledgment and Agreement

I acknowledge, understand and will comply with the above referenced security policy and rules of behavior, as applicable to my BYOD usage of ⟨Department Name⟩ services. I understand that business use may result in

```
'department' says Employee:U mustAcknowledge('this-policy')
  if Device isOwnedBy(U).
```

increases to my personal monthly service plan costs. I further understand that reimbursement of any business related data/voice plan usage of my personal device is not provided.

Appendix B

Probabilistic SecPAL Changes and Evaluation

In Chapter 6 we described how SecPAL could be modified so that assertions could carry a measure of how probable the speaker believed them to be. To implement this we would require changes to SecPAL's evaluation algorithm. Also when describing SecPAL Becker showed that it could be translated into Datalog^{C} in order to show that the language was tractable could be evaluated in polynomial time [23]. Becker et al. gave in their technical report an algorithm (5.2) translating SecPAL into Datalog^{C}. By modifying this algorithm to add the probability annotations described in this chapter, we can show the Probable SecPAL is also tractable.

The remainder of this appendix presents the modifications to SecPAL's evaluation rules, as well as Becker et al.'s Algorithm 5.2 as described by Becker with additions for probability. We do not present any arguments for the correctness of our modifications, but rather present them as a tentative first step towards implementing Probable SecPAL.

B.1 Evaluating Probability

SecPAL has three rules for evaluation: *cond*, *can-say*, and *can-act-as*. We modify the language so that the *says* keyword has an an annotation $0 \geq p \geq 1$ denoting a statements *probability*. If the annotation is missing then it is assumed to be 1 We also assume a probability combining function $\oplus$ which combines probability, though we do not define it.

$$\frac{\begin{array}{c}(A \texttt{ says } f \textit{ if } f_1 \cdots f_n \texttt{ where } c \text{ with probability at least } p_{lim}) \in \text{AC} \\ \forall i \in [1 \cdots n].\text{AC}, D \models A \texttt{ says}^{p_i} f_i\theta \qquad \vdash c\theta \quad vars(f\theta) = \emptyset \\ 0 < p_{lim} \leq \bigoplus_{i=1}^{n} p_i\end{array}}{\text{AC}, D \models A \texttt{ says}^{\bigoplus_{i=1}^{n} p_i} f\theta} \text{cond}\leq$$

$$\frac{\begin{array}{c}(A \texttt{ says } f \textit{ if } f_1 \cdots f_n \texttt{ where } c \text{ with probability is } p_{lim}) \in \text{AC} \\ \forall i \in [1 \cdots n].\text{AC}, D \models A \texttt{ says}^{p_i} f_i\theta \qquad \vdash c\theta \quad vars(f\theta) = \emptyset \\ 0 < p_{lim} \leq \bigoplus_{i=1}^{n} p_i\end{array}}{\text{AC}, D \models A \texttt{ says}^{p_{lim}} f\theta} \text{cond}=$$

$$\frac{\text{AC}, \infty \models A \texttt{ says}^{p_1} B \texttt{ can-say}_D f \quad \text{AC}, D \models B \texttt{ says}^{p_2} f}{\text{AC}, \infty \models A \texttt{ says}^{p_1 \oplus p_2} f} \text{can-say}$$

$$\frac{\text{AC}, D \models A \texttt{ says}^{p_1} x \texttt{ can-act-as } y \quad \text{AC}, D \models B \texttt{ says}^{p_2} y\, vp}{\text{AC}, D \models A \texttt{ says}^{p_1 \oplus p_2} x\, vp} \text{can-act-as}$$

$$\frac{\text{AC}, D \models A \texttt{ says}^{p'} f \quad p \leq p'}{\text{AC}, D \models A \texttt{ says}^{p} f} \text{reduce}$$

Any derived statement is at most as probable as the combination of the statements that went into deriving it.

We split the *cond* rule into two variants. The *cond*≤ rule allows us to specify a minimum probability required by combining all the conditional statements and if that limit is exceeded we take the combined probability in the probability of the outcome. The *cond*= rule allows us again to set a minimum probability but this time we take the stated probability if the rule is satisfied. These two *cond* rules serve different purposes. The *cond*= variant is useful when we want to set a limit on the probability: for instance when we have a tool with a known confidence rate we want to run, or a fact which we know how probable it is. The *cond*= rule is useful for when you want to ensure that a decision is made with a certain least-confidence, for instance if you want to be at least 80% sure that an app is safe to use before doing anything with it. In this case we would want the combined probability to trickle through the proof not the lower limit.

We also add a probability reduction rule that allows us to reduce the probability of an assertion, this allows us to phrase a policy query as *"is it at least 50% probable that..."* rather than having to discover the probabilities precisely.

B.2 A Probable Algorithm 5.2

We now describe an algorithm for translating an assertion context into an equivalent constrained Datalog program. We treat expressions of the form $e_1 says_k fact$ as Datalog literals, where k is either a variable or 0 or ∞. This can be seen as a sugared notation for a literal where the predicate name is the string concatenation of all infix operators (says, can-say, can-act-as, and predicates) occurring in the expression, including subscripts for can-say. The arguments of the literal are the collected expressions between these infix operators. For example, the expression

$$A\ says_k^p\ x\ cansay_\infty\ y\ cansay_0\ B\ canactas\ z$$

is shorthand for:

```
says_cansay_infinity_cansay_zero_canactas(A,p,k,x,y,B,z).
```

Given an assertion:

$$A \texttt{ says } f_0 \texttt{ if } f_1 \cdots f_n \texttt{ where } c \texttt{ with probability } p.$$

1. If f_0 is flat (it isn't a can-say statement), then the assertion is translated into the clause:

   A `says`$_k^{p_*}$ f_0 `:-`
   A `says`$_k^{p_1}$ f_1 $\cdots$ A `says`$_k^{p_n}$ f_n`, c,`
   p_Σ `is` $p_1 \oplus \cdots \oplus p_n$`,`
   $0 < p_{lim} \leq p_\Sigma$`.`

   Where k is a fresh variable and p_* is p_{lim} if the probability is `is`, and p_Σ is it is `at least`.

2. Otherwise f_0 is of the form:

   e_0 `can-say` D_0 $\cdots$ e_{n-1} `can-say` D_{n-1} f

   Where f is flat. Let:

   $f'_n \equiv f$ and $f'_i \equiv e_i$ `can-say` D_i f'_{i+1}, for $i \in \{0 \cdots n-1\}$.

   Note that $f_0 = f'_0$.

   Then the assertion A `says` f_0 `if` $f_1 \cdots f_m$, `c`, `with probability` p is translated into a set of $n+1$ Datalog rules as follows.

   (a) We add the Datalog rule:

   A `says`$_k^{p_*}$ f'_0 `:-`
   x `says`$_k^{p_1}$ $f_1 \cdots$ A `says`$_k^{p_m}$ f_m`, c,`
   p_Σ `is` $p_1 \oplus \cdots \oplus p_n$`,`
   $0 < p_{lim} \leq p_\Sigma$`.`

   Where k is a fresh variable, and p_* is p_{lim} if the probability is `is`, and p_Σ is it is `at least`.

   (b) For each $i \in \{1 \cdots n\}$, we add a Datalog rule

$$A \texttt{ says}_{\infty}^{p_*} f_i' \texttt{ :- } x \texttt{ says}_{D_{i-1}}^{p_1} f_i', \; A \texttt{ says}_{\infty}^{p_2} x \textit{ can-say } D_{i-1} f_i', \; p_* \texttt{ is } p_1 \oplus p_2, \; 0 < p_* \leq 1.$$

Where x is a fresh variable.

3. For each Datalog rule created above of the form:

$$A \texttt{ says}_k^{p} e\ v \texttt{ :- } \cdots$$

we add a rule:

$$A \texttt{ says}_{\infty}^{p_*} e\ v \texttt{ :- } x \texttt{ says}_k^{p_1} x \textit{ can-act-as } e, \; A \texttt{ says}_k^{p_2} e\ v, \; p_* \texttt{ is } p_1 \oplus p_2, \; 0 < p_* \leq 1.$$

Where x is a fresh variable. Note that k is not a fresh variable, but either a constant or a variable taken from the original rule.

We also add an additional rule (to account for the reduce rule) that should not be used in general, but only when trying to reduce the probability to account for a lower bound on probability in a query:

$$A \textit{ says}_k^{p_\downarrow} e\ v \texttt{ :- } A \textit{ says}_k^{p} e\ v, \; p_\downarrow \leq p.$$

Bibliography

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A Calculus for Access Control in Distributed Systems. In J. Feigenbaum, editor, *Advances in Cryptology CRYPTO 91*, number 576 in Lecture Notes in Computer Science, pages 1–23. Springer Berlin Heidelberg, 1991.

[2] M. Abadi. On SDSIs linked local name spaces. *Journal of Computer Security*, 6(1-2):3–21, 1998. ISSN 0926-227X.

[3] G.-j. Ahn, H. Hu, J. Lee, and Y. Meng. Reasoning about XACML Policy Descriptions in Answer Set Programming (Preliminary Report). In *13th International Workshop on Nonmonotonic Reasoning*, 2010.

[4] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. Awk-a pattern scanning and processing language. *Softw., Pract. Exper.*, 9(4):267–279, 1979.

[5] I. Aktug and K. Naliuka. ConSpec A Formal Language for Policy Specification. *Electronic Notes in Theoretical Computer Science*, 2008.

[6] Amazon. Amazon.co.uk Help: Amazon Appstore for Android Terms of Use. `https://www.amazon.co.uk/gp/help/customer/display.html?nodeId=201485660`, 2014.

[7] Apple. App Store Review Guidelines - Apple Developer. `https://developer.apple.com/app-store/review/guidelines/`, 2017.

[8] Apple. iTunes Connect - All Prices and Currencies - App Store. `https://itunesconnect.apple.com/WebObjects/iTunesConnect.woa/ra/ng/pricingMatrix/recurring`, 2017.

[9] Aptoide. Aptoide - Terms of Service. `https://www.aptoide.com/page/terms`, 2014.

[10] A. Armando, G. Costa, A. Merlo, L. Verderame, and K. Wrona. Developing a NATO BYOD security policy. In *International Conference on Military Communications and Information Systems*, 2016.

[11] A. Armando, G. Costa, and A. Merlo. Bring Your Own Device, Securely. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1852–1858, New York, NY, USA, 2013. ACM.

[12] A. Armando, G. Costa, A. Merlo, and L. Verderame. Enabling BYOD through secure meta-market. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2014.

[13] A. Armando, G. Costa, A. Merlo, and L. Verderame. Formal modeling and automatic enforcement of Bring Your Own Device policies. *International Journal of Information Security*, 2014.

[14] Axiomaics. Axiomatics releases free plugin for the Eclipse IDE to author XACML3.0 policies - Axiomatics. `https://www.axiomatics.com/news/axiomatics-releases-free-plugin-for-the-eclipse-ide-to-author-xacml3-0-policies/`, 2012.

[15] Axiomatics. Going on vacation, how can I implement delegation in XACML? - Axiomatics. `https://www.axiomatics.com/blog/going-on-vacation-how-can-i-implement-delegation-in-xacml/`, 2016.

[16] B. Aziz, A. Arenas, and M. Wilson. SecPAL4dsa: A Policy Language for Specifying Data Sharing Agreements. In *Secure and Trust Computing, Data Management and Applications*, number 186 in Communications in Computer and Information Science, pages 29–36. Springer, 2011.

[17] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. v. Styp-Rekowsky. AppGuard Enforcing User Requirements on Android Apps. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 543–548. Springer, Berlin, Heidelberg, 2013.

[18] D. Barrera and P. V. Oorschot. Secure Software Installation on Smartphones. *IEEE Security Privacy*, 9(3):42–48, 2011. ISSN 1540-7993.

[19] D. Barrera, J. Clark, D. McCarney, and P. C. van Oorschot. Understanding and Improving App Installation Security Mechanisms Through Empirical Analysis of Android. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 81–92, New York, NY, USA, 2012. ACM.

[20] M. Y. Becker and P. Sewell. Cassandra: distributed access control policies with tunable expressiveness. In *Proceedings. Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004. POLICY 2004.*, pages 159–168, 2004.

[21] M. Y. Becker and P. Sewell. Cassandra: flexible trust management, applied to electronic health records. In *17th IEEE Computer Security Foundations Workshop, 2004. Proceedings*, pages 139–154, 2004.

[22] M. Y. Becker. SecPAL: Formalization and Extensions. Technical Report MSR-TR-2009-127, Microsoft Research, 2009.

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

[24] M. Y. Becker, A. Malkis, and L. Bussard. A Framework for Privacy Preferences and Data-Handling Policies. Technical Report MSRTR2009128, Microsoft Research, 2009.

[25] BlackBerry. Secure Android Solution BlackBerry and Android for Work. `http://us.blackberry.com/enterprise/android-for-work.html`, 2016.

[26] A. Blass, G. De Caso, and Y. Gurevich. An introduction to DKAL. Technical Report MSR-TR-2012-108, Microsoft Research, 2012.

[27] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

[28] M. Blaze and A. D. Keromytis. RFC 2704: The keynote trust-management system version 2. Technical Report RFC 2704, IETF, 1999.

[29] M. Blaze, J. Feigenbaum, and A. D. Keromytis. KeyNote: Trust Management for Public-Key Infrastructures. In *Security Protocols*, pages 59–63. Springer, Berlin, Heidelberg, 1998.

[30] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the PolicyMaker trust management system. In *Financial Cryptography*, pages 254–274. Springer, Berlin, Heidelberg, 1998.

[31] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The Role of Trust Management in Distributed Systems Security. In J. Vitek and C. D. Jensen, editors, *Secure Internet Programming*, number 1603 in Lecture Notes in Computer Science, pages 185–210. Springer Berlin Heidelberg, 1999.

[32] G. Bruns and M. Huth. Access-Control Policies via Belnap Logic: Effective and Efficient Composition and Analysis. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 163–176, 2008.

[33] J. Bryans. Reasoning About XACML Policies Using CSP. In *Proceedings of the 2005 Workshop on Secure Web Services*, SWS '05, pages 28–35, New York, NY, USA, 2005. ACM.

[34] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9 (4), 2001.

[35] Code3PSE.org. Sample BYOD Policy. `http://www.code3pse.com/public/media/22845.pdf`, 2016.

[36] M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-Related Policy Enforcement for Android. In M. Burmester, G. Tsudik, S. Magliveras, and I. Ili, editors, *Information Security*, number 6531 in Lecture Notes in Computer Science, pages 331–345. Springer Berlin Heidelberg, 2010.

[37] G. Costantino, F. Martinelli, A. Saracino, and D. Sgandurra. Towards enforcing on-the-fly policies in BYOD environments. In *International Conference on Information Assurance and Security*, 2013.

[38] W. Dai, M. Qiu, L. Qiu, L. Chen, and A. Wu. Who Moved My Data? Privacy Protection in Smartphones. *IEEE Communications Magazine*, 55(1): 20–25, 2017. ISSN 0163-6804.

[39] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In M. Sloman, E. C. Lupu, and J. Lobo, editors, *Policies for Distributed Systems and Networks*, number 1995 in Lecture Notes in Computer Science, pages 18–38. Springer Berlin Heidelberg, 2001.

[40] A. Desnos. Androguard - Reverse engineering, Malware and goodware analysis of Android applications ... and more (ninja !). `https://github.com/androguard/androguard`, 2012.

[41] J. DeTreville. Binder, a logic-based security language. In *Proceedings 2002 IEEE Symposium on Security and Privacy*, pages 105–113, 2002.

[42] N. Dimmock, A. Belokosztolszki, D. Eyers, J. Bacon, and K. Moody. Using Trust and Risk in Role-based Access Control Policies. In *Proceedings of the Ninth ACM Symposium on Access Control Models and Technologies*, SACMAT '04, pages 156–162, New York, NY, USA, 2004. ACM.

[43] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory. Technical Report RFC 2693, IETF, 1999.

[44] W. Enck. Defending Users against Smartphone Apps: Techniques and Future Directions. In S. Jajodia and C. Mazumdar, editors, *Information Systems Security*, Lecture Notes in Computer Science, pages 49–70. Springer Berlin Heidelberg, 2011.

[45] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *16th ACM Conference on Computer and Communications Security*, pages 235–245, New York, 2009. ACM.

[46] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *USENIX Symposium on Operating Systems Design and Implementation OSDI*, October 2010.

[47] S. Fahl, M. Harbach, T. Muders, L. Baumgrtner, B. Freisleben, and M. Smith. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proceedings of the 2012 ACM Conference on Computer*

and Communications Security, CCS '12, pages 50–61, New York, NY, USA, 2012. ACM.

[48] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, New York, NY, USA, 2012. ACM.

[49] N. Fuhr. Probabilistic Dataloga logic for powerful retrieval methods. *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, 1995.

[50] Google. Android 7.1 Compatibility Definition. `https://static.googleusercontent.com/media/source.android.com/en//compatibility/7.1/android-7.1-cdd.pdf`, 2016.

[51] Google. Google Play Terms of Service. `https://play.google.com/intl/en_uk/about/play-terms.html`, 2017.

[52] N. R. C. Guerin. Security Policy for the use of handheld devices in corporate environments. Technical report, SANS, 2008.

[53] Y. Gurevich and I. Neeman. DKAL: Distributed-Knowledge Authorization Language. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 149–162, 2008.

[54] Y. Gurevich and I. Neeman. DKAL2—A Simplified and Improved Authorization Language. Technical Report MSR-TR-2009-11, Microsoft Research, 2009.

[55] J. Hallett and D. Aspinall. Towards an authorization framework for app security checking. In *Engineering Secure Software and Systems*, 2014.

[56] J. Hallett and D. Aspinall. Poster: Using Authorization Logic to Capture User Policies in Mobile Ecosystems. In *Symposium on Usable Privacy and Security*, 2015.

[57] J. Hallett and D. Aspinall. AppPAL for Android. In *8th International Symposium on Engineering Secure Software and Systems*. Springer Verlag, 2016.

[58] J. Hallett and D. Aspinall. Specifying BYOD Policies with Authorization Logic. In *PhD Symposium at iFM'16 on Formal Methods*. Reykjavik University, 2016.

[59] J. Hallett and D. Aspinall. Capturing Policies for BYOD. In *IFIP Security and Privacy Conference*, 2017.

[60] J. Hallett and D. Aspinall. Common Concerns in BYOD Policies. In *Workshop on Innovations in Mobile Privacy and Security*, 2017.

[61] J. Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46(3):311–350, 1990. ISSN 0004-3702.

[62] J. Y. Halpern and V. Weissman. Using First-Order Logic to Reason About Policies. *ACM Trans. Inf. Syst. Secur.*, 11(4):21:1–21:41, 2008. ISSN 1094-9224.

[63] H. Hao, V. Singh, and W. Du. On the effectiveness of API-level access control using bytecode rewriting in Android. *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, 2013.

[64] Harris Interactive. Privacy On and Off the Internet: What Consumers Want. Technical Report Study No. 15229, Privacy & American Business, 2002.

[65] Healthcare Information and Management Systems Society. Mobile Security Toolkit: Sample Mobile Device User Agreement. *Healthcare Information and Management Systems Society*, 2012.

[66] J. Howell and D. Kotz. A Formal Semantics for SPKI. In *Computer Security - ESORICS 2000*, pages 140–158. Springer, Berlin, Heidelberg, 2000.

[67] IBM. IBM MaaS360 - Enterprise Mobility Management (EMM). `http://www-03.ibm.com/security/mobile/maas360.html`, 2016.

[68] J.-B. Jeannin, G. d. Caso, J. Chen, Y. Gurevich, P. Naldurg, and N. Swamy. DKAL*: Constructing Executable Specifications of Authorization Protocols. In *Engineering Secure Software and Systems*, pages 139–154. Springer, Berlin, Heidelberg, 2013.

[69] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, 2012.

[70] X. Jin, L. Wang, T. Luo, and W. Du. Fine-Grained Access Control for HTML5-Based Mobile Applications in Android. In *Information Security*, pages 309–318. Springer, Cham, 2015.

[71] Joseph Y. Halpern and Ron van der Meyden. A Logic for SDSI's Linked Local Name Spaces. *Journal of Computer Security*, 1999.

[72] A. Jsang and S. L. Presti. Analysing the Relationship between Risk and Trust. In C. Jensen, S. Poslad, and T. Dimitrakos, editors, *Trust Management*, number 2995 in Lecture Notes in Computer Science, pages 135–145. Springer Berlin Heidelberg, 2004.

[73] G. Kennington, K. Pointer, C. Morey, L. Budge, V. Dunn, and S. Ball. Mobiles Devices Policy. Technical report, Torbay and Southern Devon Health and Care NHS Trust, 2014.

[74] A. Levy and Y. Sagiv. Constraints and redundancy in datalog. *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1992.

[75] A. Levy, I. S. Mumick, Y. Sagiv, and O. Shmueli. Equivalence, query-reachability and satisfiability in Datalog extensions. *Proceeding of the 12th ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems*, 1993.

[76] N. Li and J. C. Mitchell. Datalog with Constraints: A Foundation for Trust Management Languages. In V. Dahl and P. Wadler, editors, *Practical Aspects of Declarative Languages*, number 2562 in Lecture Notes in Computer Science, pages 58–73. Springer Berlin Heidelberg, 2003.

[77] N. Li, B. Grosof, and J. Feigenbaum. A Practically Implementable and Tractable Delegation Logic. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S P 2000*, pages 27–42, 2000.

[78] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *Proceedings 2002 IEEE Symposium on Security and Privacy*, pages 114–130, 2002.

[79] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *Proceedings 2002 IEEE Symposium on Security and Privacy*, pages 114–130, 2002.

[80] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation Logic: A Logic-based Approach to Distributed Authorization. *ACM Trans. Inf. Syst. Secur.*, 6(1): 128–171, 2003. ISSN 1094-9224.

[81] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management*. *Journal of Computer Security*, 11(1): 35–86, 2003. ISSN 0926-227X.

[82] J. Lin, B. Liu, N. Sadeh, and J. I. Hong. Modeling Users Mobile App Privacy Preferences: Restoring Usability in a Sea of Permission Settings. *Symposium On Usable Privacy and Security*, 2014.

[83] F. Martinelli, P. Mori, and A. Saracino. Enhancing Android Permission Through Usage Control: A BYOD Use-case. In *Symposium on Applied Computing*, 2016.

[84] MobileIron Security Labs. Q4 Mobile Security and Risk Review. Technical report, MobileIron Security Labs, 2015.

[85] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM.

[86] H. Nergaard, N. Ulltveit-Moe, and Terje Gjøsæter. A scratch-based graphical policy editor for XACML. *International Conference on Information Systems Security and Privacy*, 2015.

[87] OASIS. XACML v3.0 Administration and Delegation Profile. Technical report, OASIS, 2010.

[88] OASIS. eXtensible Access Control Markup Language (XACML) Version 3.0. Technical report, OASIS, 2013.

[89] OASIS XACML Technical Comitee. Abbreviated Language for Authorization. Technical report, OASIS, 2015.

[90] A. J. Oliner, A. P. Iyer, I. Stoica, E. Lagerspetz, and S. Tarkoma. Carat: Collaborative Energy Diagnosis for Mobile Devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 10:1–10:14, New York, NY, USA, 2013. ACM.

[91] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. *Security and Communication Networks*, 5(6):658–673, 2012. ISSN 1939-0122.

[92] A. N. Prior. Past, Present and Future. Technical report, Oxford University, 1967.

[93] R. Sikoryak. *Terms and Conditions*. Drawn and Quarterly, 01 edition edition, 2017.

[94] C. D. P. K. Ramli. Detecting Incompleteness, Conflicting and Unreachability XACML Policies using Answer Set Programming. *arXiv:1503.02732 [cs]*, 2015.

[95] C. D. P. K. Ramli, H. R. Nielson, and F. Nielson. XACML 3.0 in Answer Set Programming. In E. Albert, editor, *Logic-Based Program Synthesis and Transformation*, number 7844 in Lecture Notes in Computer Science, pages 89–105. Springer Berlin Heidelberg, 2012.

[96] C. D. P. K. Ramli, H. R. Nielson, and F. Nielson. The logic of XACML. *Science of Computer Programming*, 83:80–105, 2014. ISSN 0167-6423.

[97] M. Research. SecPAL Research Release for Microsoft .NET, version 1.1. `https://www.microsoft.com/en-us/download/details.aspx?id=52356`, 2007.

[98] P. Z. Revesz. Constraint databases: A survey. In *Semantics in Databases*, pages 209–246. Springer, Berlin, Heidelberg, 1995.

[99] P. Z. Revesz. Safe Datalog Queries with Linear Constraints. In *Principles and Practice of Constraint Programming CP98*, pages 355–369. Springer, Berlin, Heidelberg, 1998.

[100] F. Salim, J. Reid, E. Dawson, and U. Dulleck. An Approach to Access Control under Uncertainty. In *2011 Sixth International Conference on Availability, Reliability and Security (ARES)*, pages 1–8, 2011.

[101] H. Schulze. BYOD & Mobile Security 2016 Spotlight Report. Technical report, LinkedIn Information Security, 2016.

[102] R. Smith, B. Taylor, C. Silva, M. Bhat, T. Cosgrove, and J. Girard. Magic Quadrant for Enterprise Mobility Management Suites. Technical Report G00279887, Gartrer, 2016.

[103] Solar Designer. John the Ripper password cracker. `http://www.openwall.com/john/`, 2013.

[104] V. Svajcer and S. McDonald. Classifying PUAs In The Mobile Environment. In *Virus Bulletin Conference*, 2013.

[105] C. Thompson, M. Johnson, S. Egelman, D. Wagner, and J. King. When It's Better to Ask Forgiveness Than Get Permission: Attribution Mechanisms for Smartphone Resources. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, SOUPS '13, pages 1:1–1:14, New York, NY, USA, 2013. ACM.

[106] K. Thompson. Reflections on Trusting Trust. *Turing Award Lecture*, 1984.

[107] H. T. T. Truong, E. Lagerspetz, P. Nurmi, A. J. Oliner, S. Tarkoma, N. Asokan, and S. Bhattacharya. The Company You Keep: Mobile Malware Infection Rates and Inexpensive Risk Indicators. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 39–50, New York, NY, USA, 2014. ACM.

[108] M. C. Tschantz and S. Krishnamurthi. Towards reasonability properties for access-control policy languages. In *ACM Symposium on Access Control Models and Technologies*, 2006.

[109] K. Twidle, N. Dulay, E. Lupu, and M. Sloman. Ponder2: A Policy System for Autonomous Pervasive Environments. In *2009 Fifth International Conference on Autonomic and Autonomous Systems*, 2009.

[110] J. M. Urban and C. J. Hoofnagle. The privacy pragmatic as privacy vulnerable. *Symposium on Usable Privacy and Security*, 2014.

[111] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Permission evolution in the android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 31–40. ACM, 2012.

[112] D. Williamson, A. Grzybowski, and S. Graham. Bring Your Own Device Policy. Policy 15, University of Edinburgh, 2015.

[113] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos Operating System. *ACM Trans. Comput. Syst.*, 12(1):3–32, 1994. ISSN 0734-2071.

[114] R. Xu, H. Sadi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *21st USENIX Security Symposium*, Bellevue, WA, 2012. USENIX Association.

[115] Yandex. YANDEX.STORE TERMS OF USE Legal documents. `http://yandex.com/legal/store_termsofuse/index.html`, 2014.

[116] N. Zhang, M. Ryan, and D. P. Guelev. Synthesising Verified Access Control Systems in XACML. In *Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering*, FMSE '04, pages 56–65, New York, NY, USA, 2004. ACM.