Ch 6
Draft 1.

Too much for a regular "Future work" Chapter. Either put into another chap

Authorization Logic For Mobile Ecosystems

"Towards AppPAL applications and extensions"

or cut content here to core ideas only (not giving enough to be shooting yourself in the foot with vulnerabilities, errors)

Joseph Hallett

Doctor of Philosophy

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

2017

But You've made significant steps here. I would expect Future Work chap to be much shorter, ~5 pages.

Chapter 6

Future Work

Throughout this thesis we have described the policies of the mobile ecosystem and how AppPAL can be used 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. This chapter looks at two areas in particular and describes our *preliminary* explorations. We describe how AppPAL policies can be analyzed automatically and describe an early prototype to check policies for common problems. We also describe preliminary thoughts about how plausibility, quantified doubt, could be used to express confidence in a statement.

6.1 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?

Many authorization and logic-programming languages, such as XACML [13] and Datalog [1, 2], have thought about how policies can be checked for various properties similar to those described here. The properties we seek to check

researchers thoughts, not XACML

AppPAL (or SecPAL) for are not novel, but rather the application of them to SecPAL is.

We have implemented preliminary checkers for some of these properties, though they are undertested. The code is included with AppPAL.

6.1.1 Checking Satisfiability

If an assertion is *satisfiable* then there are sufficient facts such that the assertion's conditions could be met. 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.

When an assertion is satisfiable there is a combination of facts that could satisfy its conditionals. If there are no facts that could satisfy the rule then the policy may be unsatisfiable as there are rules that can never be used. For any goal G:

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

This is a similar idea to the notions of *satisfiability* in Datalog (and more generally logic programming). Satisfiability in Datalog is defined as [1]:

> **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.

For an example of satisfiability, consider the following snippet taken from the NHS policy described in **??**. The rule described in the policy is that an app must be approved by the Integrated Governance Comitee (IGC) as well as by either the Care and Clinical Policies Group (CACPG) as well as the Management of Information Group (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 we run the satisfiability checker on this policy it reports that (amongst 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.

Using AppPAL we can describe policies using delegation. As well as looking for assertions that are unsatisfiable it also looks for examples where we have a *can-say* statement, but lack any statements from the delegated party to that effect. This may indicate that we need to collect additional statements from the delegated person, or that the *can-say* statement can be removed as it will not be used. This check is somewhat 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. When we run the satisfiability checker however, 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 be made.

Yes. Expand this to explain better. I think you should be clever about variables.

6.1.2 Checking Redundancy

This is great, but you just need a bit more precision about what it does, soundness/completeness questions (Ok if not S or C, but you need to say)

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 [2] that we care about here:

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

maybe separate into 2 subsecs

These ideas are directly relatable to AppPAL, for instance if we have 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

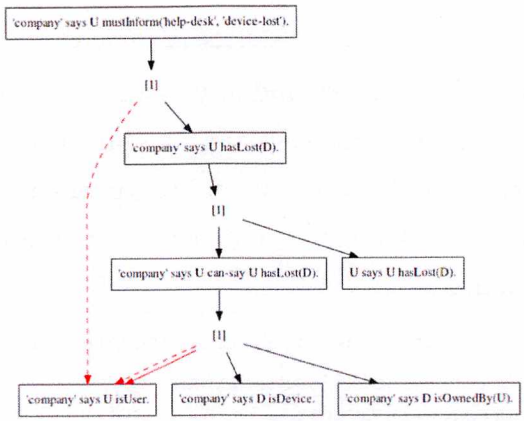

Figure 6.1: Proof graph showing irrelevance.

occurs when using the typed-syntax described in **??**. 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 check that *U* is a user but it is irrelevant as the check had already been done when checking is *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).
```

We can check for this kind of 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 6.1: the irrelevant links are shown in red and the AppPAL facts connected to the two dashed ones can be removed to remove the irrelevance.

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:

Listing 6.1: Output of AppPAL when checking a policy with unreachability.

```
java -jar Lint.jar --redundancy unreachable.policy
[I]: loaded 1/1 files of 2 assertions
[I]: flattened 0 node(s)
[W]: 'alice' says App isInstallable. has unreachable derivation trees.
```

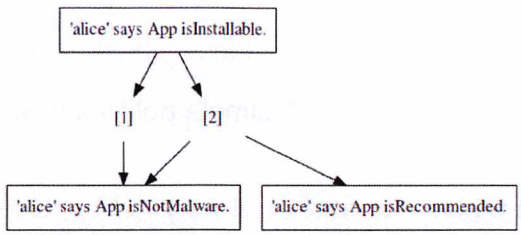

Figure 6.2: Proof graph showing unreachability.

```
'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 6.2). For each proof node we collect the facts (the leaves of the graph underneath it, which are the ground assertions from the AC[1] 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 it contains facts which are not in the minimal derivation tree. In the case of Figure 6.2, 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 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

[1]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.
```

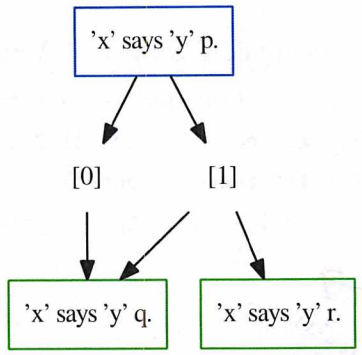

Figure 6.3: A simple policy shown as a graph.

also be, but when B is satisfied A may not be: consequently A is redundant as B can be used to prove its goal with fewer facts. For any goal G.

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

Additionally if two different goals (G and G') have equivalent proofs, 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 6.3. 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 6.4 we can see it is more complex. Goals that depend on more than just green facts, are shown as black 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 6.4). 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).

6.2 Plausible AppPAL

The SecPAL authorization language, and the AppPAL instantiation, allow 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 but not absolutely certain.

With current authorization languages, such as AppPAL and XACML, there is no way to quantify the belief a principal has in any statement. A principal cannot say how plausable they think any statement is.

6.2.1 Examples of Plausability

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

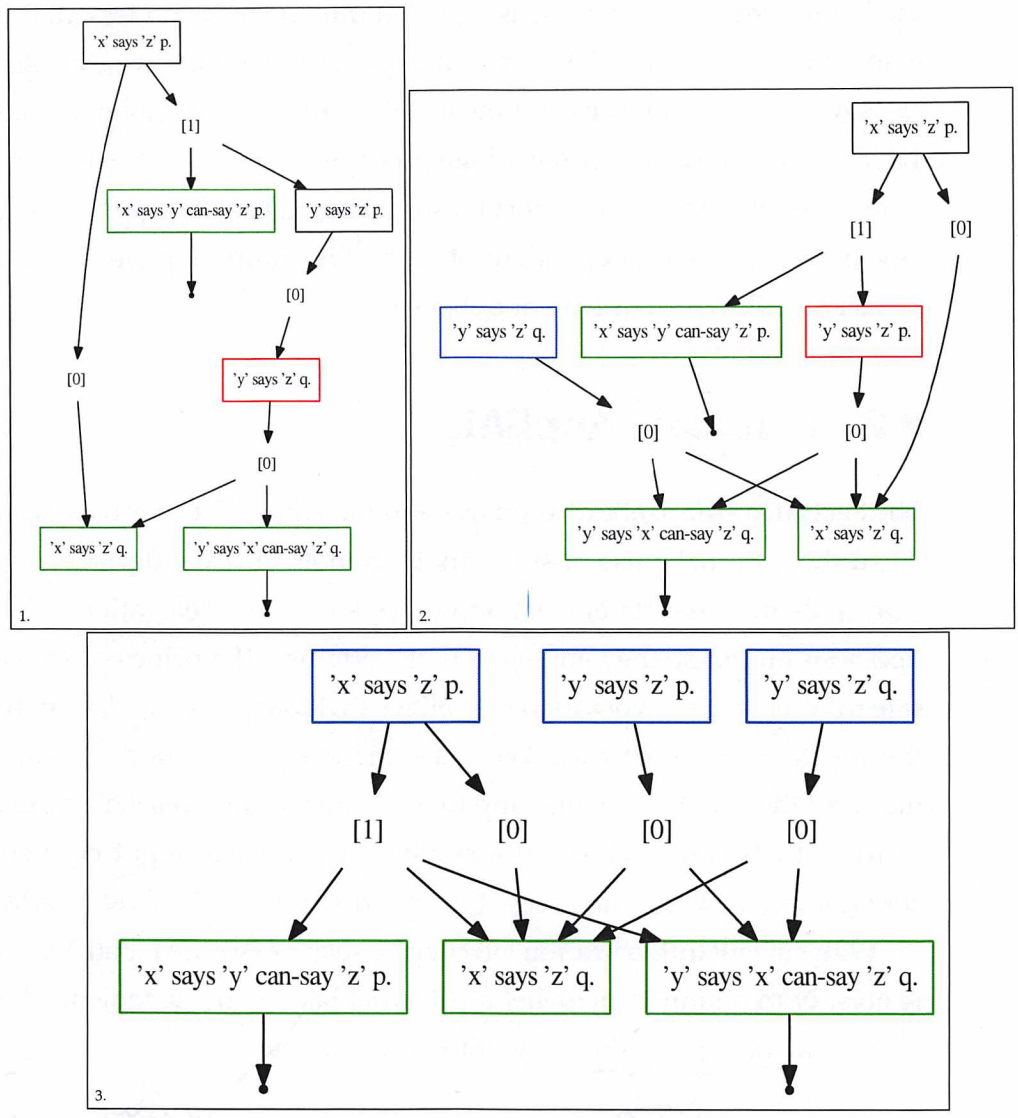

Figure 6.4: Flattening a more complex policy.

not.

AppPAL, however, is primarily for deciding what apps you want to use. Whether you want to install an app or not is less binary than an access control decision, because it is ultimately an opinion. Consider a really simple policy that says *"do not install malware"*: you could try using a malware scanner to check apps, but there opinions on apps can change rapidly and often. You could use a meta-scanning tool like *VirusTotal*, but then you'd only get the percentage of antivirus tools that flagged the app as malicious (Figure 6.5). The scan in Figure 6.5 was for a sample of the *BaseBridge* malware, taken from the *Android Malware Genome Project* [15]. Even for this relatively old malware sample (from 2012) 15 antivirus programs, including McAfee and Yandex's programs, did not flag it as dangerous. In contrast a scan of the *towelroot* app [7], an app which will grant root access to a device but is not *in itself* malicious, 3 out of 64 antivirus checkers reported the download source as dangerous. If the app were to be installed on a device then a further warning would be displayed by Google's own built-in antivirus. This last app is not technically malicious, but it is certainly dangerous. The assertion that an *app is safe* is not a binary decision. Without plausability we cannot represent the doubt and confidence in any assertion.

For another example consider a policy you only want to install apps that are made by *reputable* developers and that are *safe*. Both reputable and safe are poorly defined, and badly represented by a binary decision. A developer may be reputable if they are a large developer with a lot of staff like Google or Facebook, they might be reputable if their apps have been well reviewed, but what about a developer who produces a large number of games with in-app-purchases, TV adverts, and sketchy privacy records? They are probably more reputable than a malware author but you might have less confidence that they are producing good apps than Google. An app might be seen as safe if it doesn't request any permissions and has no native code, but if it starts requesting more permissions and the amount of native code grows then a user's confidence that it is safe might fall. When combined into the policy as a whole Google may be able to get away with a lot more permissions than other developers simply because users trust them more not to be evil. Again, without plausability it is difficult to represent these decisions.

Probabilistic versions of Datalog are not a new idea. Various papers pro-

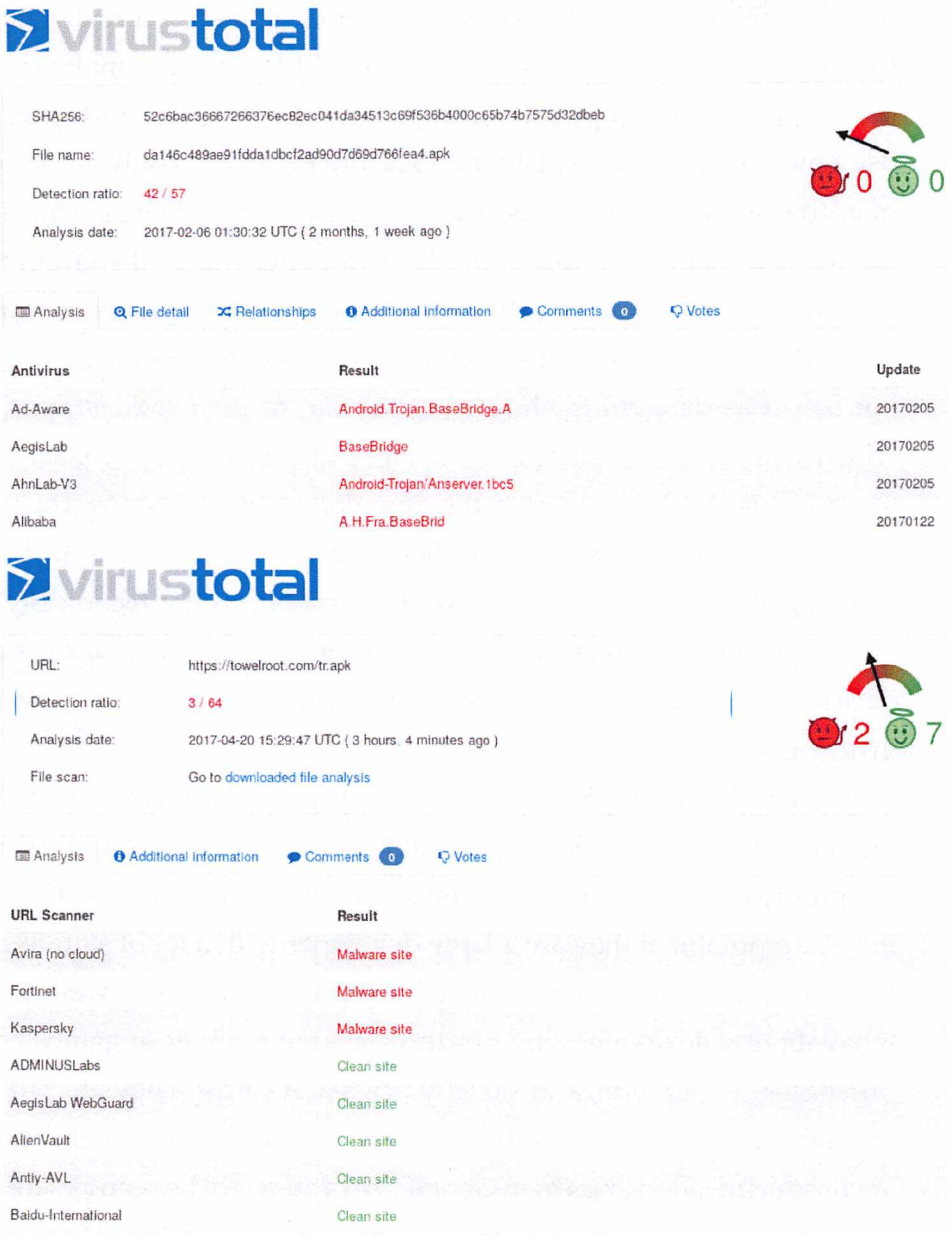

Figure 6.5: VirusTotal results for 57 antivirus packages scanning a sample of the BaseBridge Android malware, and 64 antivirus packages analyzing a download of the Towelroot rootkit.

posed probabilistic variants of Datalog [6] or explored the semantics of probabilistic logics [9]. Role-based access control languages have incorporated ideas about risk into their schemes [10, 5, 14], which is a similar notion to plausability 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. Part of the work toward this this year has included modifying Becker's evaluation and translation-to-Datalog algorithms [3] to include a plausability value, and for giving rules to query on the basis of them, using Datalog^C [11].

6.2.2 Modifying AppPAL for Plausability

AppPAL 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 *plausability*. If the annotation is missing then it is assumed to be 1 We also assume a plausability combining function $\oplus$ which combines plausability. The AppPAL inference rules then become as follows, with additions highlighted in red.

$$\frac{(A \text{ says } f \text{ if } f_1 \cdots f_n \text{ where } c \text{ with plausability at least } p_{lim}) \in \text{AC} \quad \forall i \in [1 \cdots n].\text{AC}, D \models A \text{ says}^{p_i} f_i\theta \quad \vdash c\theta \quad vars(f\theta) = \emptyset \quad 0 < p_{lim} \leq \bigoplus_{i=1}^{n} p_i}{\text{AC}, D \models A \text{ says}^{\bigoplus_{i=1}^{n} p_i} f\theta} \text{ cond}\leq$$

$$\frac{(A \text{ says } f \text{ if } f_1 \cdots f_n \text{ where } c \text{ with plausability is } p_{lim}) \in \text{AC} \quad \forall i \in [1 \cdots n].\text{AC}, D \models A \text{ says}^{p_i} f_i\theta \quad \vdash c\theta \quad vars(f\theta) = \emptyset \quad 0 < p_{lim} \leq \bigoplus_{i=1}^{n} p_i}{\text{AC}, D \models A \text{ says}^{p_{lim}} f\theta} \text{ cond}=$$

$$\frac{\text{AC}, \infty \models A \text{ says}^{p_1} B \text{ can-say}_D f \quad \text{AC}, D \models B \text{ says}^{p_2} f}{\text{AC}, \infty \models A \text{ says}^{p_1 \oplus p_2} f} \text{ can-say}$$

$$\frac{\text{AC}, D \models A \text{ says}^{p_1} x \text{ can-act-as } y \quad \text{AC}, D \models B \text{ says}^{p_2} y\ vp}{\text{AC}, D \models A \text{ says}^{p_1 \oplus p_2} x\ vp} \text{ can-act-as}$$

$$\frac{\text{AC}, D \models A \text{ says}^{p'} f \quad p \leq p'}{\text{AC}, D \models A \text{ says}^{p} f} \text{ reduce}$$

In general any derived statement is at most as plausible as the combination of the statements that went into deriving it. We split the *cond* rule into two variants. The *cond*$\leq$ rule allows us to specify a minimum plausability required

by combining all the conditional statements and if that limit is exceeded we take the combined probability in the plausability of the outcome. The *cond=* rule allows us again to set a minimum plausability but this time we take the stated plausability 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 plausability: for instance when we have a tool with a known confidence rate we want to run, or a fact which we know how plausible 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 plausability to trickle through the proof not the lower limit.

We also add a plausability reduction rule that allows us to reduce the plausability of an assertion, this allows us to phrase a policy query as *"is it at least 50% plausable that..."* rather than having to discover the plausabilities precisely.

6.2.2.1 Plausability Combination

How should the plausability combination operator be defined? One simplistic approach might be to take the least plauable assertion as the final value. This however is too simple. Consider the case where an app needs to be safe and from a reputable developer. Say we're 50% sure the app is safe but we are unsure of the developer. We are 90% sure Google is reputable and only 60% sure Rovio is. No matter who the developer is the combined probability is the same (50%) so the distinction is lost.

$$p_a \oplus_{\min} p_b \leftarrow \min p_a\, p_b$$

If we are sure the plausabilities are independent we could multiply them together to get the final plausability. This is appealing in that if we take the example above if Google is the developer we're 45% sure the app is usable, but only 30% if Rovio is. If however there are a lot of combinations to be made the probabilities may get very small and be difficult to distinguish.

$$p_a \oplus_{\text{ind}} p_b \leftarrow p_a \times p_b$$

Both the two previous suggestions are naïve in their approach. The first

is unnecessarily conservative, the second assumes independence. A better scheme may be to incorporate a more sophisticated reasoning system. One system might be Dempster-Schafer theory [4, 8] which is designed to reason about the probability of provability of any statement. It can also be used to give a value to how close evidence comes to rendering a proposition provable [12]. Other schemes, and working out how to implement and integrate this into AppPAL is left to future work.

6.2.3 Guarantees for Plausible AppPAL

However plausibility combination is implemented some thought should be given to what properties and guarantees AppPAL ought to offer. One guarantee might be that if all statements are perfectly plausible then this ought to be equivalent to standard AppPAL. Another might be that if a statement is perfectly implausible then it ought to be equivalent to the statement not existing at all. A third rule should be that we can't make a statement more plausible by combining less plausible statements. To summarize:

- If all statements have a plausability of 1, then this is equivalent to standard AppPAL.
- If a statement has a plausability of 0, then it is equivalent to the statement not existing in the assertion context.
- No statement should be more certain than the conditionals used to derive it.

Point 1 is true since if $p_{\text{lim}} = 1$ then no statement will be accepted unless its plausability is also 1. If we combine the plausability of two events this will also be 1 (using either method described so far): hence the restriction on the plausability in the cond rule is always true since all statements have a plausability of one. Rewriting the rules with this in mind we get the original AppPAL inference rules, so if all statements have a plausability of 1, then this must be equivalent to standard AppPAL.

Point 2 is also true by the restriction on probabilities in the cond rule. Since p_{lim} must be greater than 0, if a statement has a plausability of 0 then it will not be accepted. Similarly when combining plausabilities if a statement is

completely inplausible then the plausibility of it happening with another event must also be implausible. Hence the combination should always be 0, and no statement combined with an implausible statement will be derivable.

Point 3 is important as we don't want a means to make a statement more certain by repeatedly applying a rule. For example imagine we had a combination operator that took the sum (or 1 if the combination was greater than 1), and a rule such as:

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

If we also have statement that x says$^{0.2}$ yp, then we could apply this rule to get x says$^{0.4}$ yp, and so on until we're certain.

Similarly if we had statements such as:

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

If we knew, with certainty, that `'x' says 'z' p`, then if we used a plausibility combination function that averaged the plausibilities we would increase the overall plausibility that `'x' says 'y' p`.

The multiplication and minimum plausability rules won't allow you to increase plausability like this (minimum is trivial: if it is always the plausability of the least plausible statement then plausabilities will never increase; multiplication works because if all plausabilities are between 1 and 0, then the product will also never increase).

6.2.4 Worked Examples

6.2.4.1 Tools with Confidences

Suppose we have a static analysis tool with a false positive rate of 10%, (i.e. one app in ten it will falsely flag as having issues when infact it is fine).

```
'user' says App isSafe
  if App isAnApp
  where staticAnalyisTool(App) = true
  with plausability is 0.9.
```

This is useful because we might want to treat static analysis results, along with other information, as more reliable. For instance we might say an app

which isn't malware is good, but an app that isn't malware and is recommended by a friend is even better!

```
'user' says App isGood
  if App isntMalware
  with plausability is 0.5.

'user' says app isGood
  if App isntMalware,
     App isRecommended
  with plausability is 0.8.
```

6.2.4.2 Review scores

When picking games we may rely on expert reviewers to suggest apps to us. It is very common to give reviewed items a score at the end of the review.

```
'less-trusted-reviewer' says 'angry-birds' isGood
  with plausability is 0.8.

'trusted-reviewer' says 'angry-birds' isGood
  with plausability is 0.9.
```

We might also trust some reviewers more than others, again we can express this screnario with plausability and attempt to normalize their scores somewhat.

```
'user' says 'trusted-reviewer' can-say App isGood
  if App isAnApp
  with plausability is 1.0.

'user' says 'less-trusted-reviewer' can-say App isGood
  if App isAnApp
  with plausability is 0.5.
```

When we come to picking an app though we might like to rely on multiple bits of information however.

```
'user' says App isInstallable
  if App isGood,
     App isSafe
  with plausability at least 0.75.
```

6.2.5 Translation to Datalog^C

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 [3]. Becker et al. gave in their technical report an algorithm (5.2) translating SecPAL into Datalog^C. By modifying this algorithm to add the plausibility annotations described in this chapter, we can show the Plausible AppPAL is also tractable. The remainder of this chapter presents Becker et al.'s Algorithm 5.2 as described by Becker with additions for plausability.

6.2.5.1 A Plausible 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

$$A\ says^{p}_{k}\ x\ cansay_{\infty}\ y\ cansay_{0}\ B\ canactas\ z$$

is shorthand for:

```
says_cansay_infinity_cansay_zero_canactas(A,p,k,x,y,B,z).
```

Given an assertion:

$$A\ \texttt{says}\ f_0\ \texttt{if}\ f_1 \cdots f_n\ \texttt{where}\ c\ \texttt{with plausability}\ p.$$

1. If f_0 is flat (it isn't a can-say statement), then the assertion is translated into the clause:

$$\begin{array}{l} A\ \texttt{says}^{p_*}_{k}\ f_0\ \texttt{:-} \\ \quad A\ \texttt{says}^{p_1}_{k}\ f_1 \cdots A\ \texttt{says}^{p_n}_{k}\ f_n,\ c, \\ \quad p_\Sigma\ \texttt{is}\ p_1 \oplus \cdots \oplus p_n, \\ \quad 0 < p_{lim} \leq p_\Sigma. \end{array}$$

   Where k is a fresh variable and p_* is p_{lim} if the plausability is `is`, and p_Σ is it is `at least`.

2. Otherwise f_0 is of the form:

   $e_0\ \textit{can-say}\ D_0 \cdots e_{n-1}\ \textit{can-say}\ D_{n-1}\ f$

   Where f is flat. Let:

   $f'_n \equiv f$ and $f'_i \equiv e_i\ \textit{can-say}\ D_i\ f'_{i+1}$, 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 plausability` p is translated into a set of $n+1$ Datalog rules as follows.

(a) We add the Datalog rule:

$$\begin{array}{l} A \text{ says}^{p_*}_k \ f'_0 \ \texttt{:-} \\ \quad x \text{ says}^{p_1}_k \ f_1 \cdots A \text{ says}^{p_n}_k \ f_m, \ \texttt{c}, \\ \quad p_\Sigma \ \texttt{is} \ p_1 \oplus \cdots \oplus p_n, \\ \quad 0 < p_{lim} \leq p_\Sigma. \end{array}$$

Where k is a fresh variable, and p_* is p_{lim} if the plausability is `is`, and p_Σ is it is `at least`.

(b) For each $i \in \{1 \cdots n\}$, we add a Datalog rule

$$\begin{array}{l} A \text{ says}^{p_*}_\infty \ f'_i \ \texttt{:-} \\ \quad x \text{ says}^{p_1}_{D_{i-1}} \ f'_i, \\ \quad A \text{ says}^{p_2}_\infty \ x \ \textit{can-say} \ D_{i-1} \ f'_i, \\ \quad p_* \ \texttt{is} \ p_1 \oplus p_2, \\ \quad 0 < p_* \leq 1. \end{array}$$

Where x is a fresh variable.

3. For each Datalog rule created above of the form:

$$A \text{ says}^{p}_k \ e \ v \ \texttt{:-} \cdots$$

we add a rule:

$$\begin{array}{l} A \text{ says}^{p_*}_\infty \ e \ v \ \texttt{:-} \\ \quad x \text{ says}^{p_1}_k \ x \ \textit{can-act-as} \ e, \\ \quad A \text{ says}^{p_2}_k \ e \ v, \\ \quad p_* \ \texttt{is} \ p_1 \oplus p_2, \\ \quad 0 < p_* \leq 1. \end{array}$$

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 plausability to account for a lower bound on plausability in a query:

$$\begin{array}{l} A \text{ says}^{p_\downarrow}_k \ e \ v \ \texttt{:-} \\ \quad A \text{ says}^{p}_k \ e \ v, \\ \quad p_\downarrow \leq p. \end{array}$$

Bibliography

[1] Alon Levy, Inderpal Singh Mumick, Yehoshua Sagiv, and Oded Shmueli. Equivalence, query-reachability and satisfiability in Datalog extensions. *Proceeding of the 12th ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems*, 1993.

[2] Alon Levy and Yehoshua Sagiv. Constraints and redundancy in datalog. *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1992.

[3] Moritz Y. Becker, Cdric Fournet, and Andrew D. Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Journal of Computer Security*, January 2010.

[4] A. P. Dempster. Upper and Lower Probabilities Induced by a Multivalued Mapping. *The Annals of Mathematical Statistics*, 38(2):325–339, 1967.

[5] Nathan Dimmock, Andrs Belokosztolszki, David Eyers, Jean Bacon, and Ken 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.

[6] Norbert 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.

[7] George (geohot) Hotz. towelroot by geohot.

[8] Glenn Shafer. *A Mathematical Theory of Evidence*. Princeton University Press, 1976.

[9] Joseph Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46(3):311–350, December 1990.

[10] Audun Jsang and Stphane Lo Presti. Analysing the Relationship between Risk and Trust. In Christian Jensen, Stefan Poslad, and Theo Dimitrakos, editors, *Trust Management*, number 2995 in Lecture Notes in Computer Science, pages 135–145. Springer Berlin Heidelberg, March 2004. DOI: 10.1007/978-3-540-24747-0_11.