Information Flow in Trust Management Systems

Moritz Y. Becker

Microsoft Research

7 JJ Thomson Avenue,

Cambridge, United Kingdom

moritzb@microsoft.com

Tel.: +44 1223 479826

Keywords: Information flow, access control, trust management, policy language, Datalog, probing attack, opacity

Abstract

This article proposes a systematic study of information flow in credential-based declarative authorization policies. It argues that a treatment in terms of information flow is needed to adequately describe, analyze and mitigate a class of probing attacks which allow an adversary to infer any confidential fact within a policy. Two information flow properties that have been studied in the context of state transition systems, non-interference and opacity, are reformulated in the current context of policy languages. A comparison between these properties reveals that opacity is the more useful, and more general of the two; indeed, it is shown that non-interference can be stated in terms of opacity. The article then presents an inference system for non-opacity, or detectability, in Datalog-based policies. Finally, a pragmatic method is presented, based on a mild modification of the mechanics of delegation, for preventing a particularly dangerous kind of probing attack that abuses delegation of authority.

1 Introduction

This article is motivated by a class of attacks on access control systems. The systems that are susceptible to such attacks are the ones that subscribe to the trust management principle [9]: authorization decisions are based on the service's local policy (consisting of declarative assertions) in union with a (potentially empty) set of supporting credentials (containing assertions) submitted by the requester. Access requests are mapped to queries, so that access is granted if the corresponding query succeeds when evaluated in the context of the policy and the submitted credentials.

It turns out that if the language supports credentials of sufficient expressiveness, then by creating and submitting certain credentials and by observing the corresponding results of a series of access requests, an adversary can potentially gain knowledge about any confidential fact in the service's local policy. Such probing attacks are particularly dangerous when mounted against policies written in languages supporting decentralized delegation of authority (e.g. Delegation Logic [29], SPKI/SDSI [13], RT [31], SD3 [27], Binder [16], Cassandra [8], SecPAL [5]). Gurevich and Neeman [23] describe an example of such an attack on SecPAL. We briefly recall their example (in a paraphrased, but equivalent form). Imagine a service with a policy that gives parking permissions to principals that consent to the relevant terms and conditions, but additionally contains confidential facts (here hyperbolically represented by secret agent memberships). The adversary Alice submits two self-issued credentials containing the following assertions to the service, together with her request for a parking permission:

(1) Alice says Alice consents to parking rules if
    Bob is a secret agent

(2) Alice says Service can say
    Bob is a secret agent

Suppose the access request succeeds: in other words, the corresponding query ⟨Service says Alice can park⟩ evaluates to true in the service's policy augmented by (1) and (2), and this positive result is observed by Alice. Gurevich and Neeman [23] argue that Alice can now

infer a secret from this observation, namely that (3) Service says that Bob is a secret agent: (3) and (2) together would imply that Alice also says that Bob is a secret agent, hence the condition of (1) would be satisfied and Alice's consent would be valid, which in turn would explain why the query succeeds. However, a closer look reveals that Alice cannot be absolutely certain that (3) holds in the service's policy. For instance, it may be the case that (3) is not true, but the fact ⟨Alice says Bob is a secret agent⟩ is true in the policy, which would result in the same observation from Alice's restricted point of view.

But the attack can be made more precise by Alice conducting a second *probe*, again with the same request (and thus the same query), but submitting only the first of the two assertions above. If this second probe yields a negative response, Alice can be certain that Bob is indeed a secret agent (according to Service), for then she knows that (2) is essential in making the query succeed. A similar attack can be mounted in order to check for the *absence* of any secret agent fact. Such attacks, where successive probing effectively provides a covert channel for confidential facts, work with a wide range of modern policy systems, including the ones mentioned above, as well as any of their derivatives.

Gurevich and Neeman propose a new policy language, DKAL [23], with a "says to" construct, to mitigate such unintended information leaks. However, as we will argue towards the end of this article, their solution does not offer a good tradeoff between protection, expressiveness and usability. But more fundamentally, it lacks a formal framework that could state which security properties an effective mitigation mechanism should guarantee. There has so far been no systematic description of what is meant by "probing attack", and no analysis on what information is leaked. Indeed, as we demonstrate in Section 2, DKAL2 (DKAL's successor) [24], is prone to probing attacks itself. Moreover, as the example above shows, informal reasoning easily leads to false inferences, and later examples in this article will illustrate that the general inference problem is far from trivial.

The problem is clearly closely related to the well-studied research area of information flow. However, information flow in credential-based policy systems is distinctly different from that in state transition systems and executable programs, which have traditionally been the focus of such research. In particular, in the current setting there is no notion of state, state transition,

or trace, and, most importantly, adversaries have the power to inject credentials into the policy. As such, the established techniques cannot be directly transferred to credential-based policy systems.

We wish to initiate a systematic study of information flow in the context of credential policies. The current article is a first step in this direction. Our main contributions can be summarized as follows.

- A reformulation of the well-known information flow property, *non-interference* [21], in the context of credential policies, based on observational equivalence. We then consider another property, *opacity* [11], which so far has received somewhat less attention. We redefine opacity as the inability to infer a specific predicate about a policy and prove that non-interference can be stated in terms of opacity. Opacity turns out to be far more useful (but harder to check) for our purposes. (Section 3)

- An inference system for checking the negation of opacity, which we call *detectability*, in policies based on Datalog. The system can be used to analyze probing attacks and to prove that certain properties about a policy are leaked. (Section 4)

- A pragmatic method for mitigating information leaks caused by delegation-based probing. The method consists of a mild modification of the delegation mechanism in policy languages. The concepts developed in this article allow us to precisely describe the security guarantees from this modification. (Section 5)

Connections to traditional information flow, the database inference problem, automated trust negotiation, and hypothetical logic programming are pointed out in Section 6. We discuss further issues and conclude with a critical examination of alternative methods for preventing probing attacks in Section 7. An earlier, abridged version of this article appeared in a conference proceedings [4].

2 Probing attacks in DKAL2

Gurevich and Neeman [23] argue that credentials containing conditions that are evaluated on the service's site, as commonly supported in logic-based policy languages such as SecPAL, enable adversaries to gain confidential information via probing attacks. Section 1 shows a simple example of such an attack in SecPAL.

DKAL [23] is a language specifically designed to prevent probing attacks, but the paper does not specify exactly what it is trying to protect against and how, as it lacks a precise framework for reasoning about probing attacks. At first sight, the mechanism that is supposed to provide protection is a "to" operator, which replaces SecPAL's says keyword, and specifies and restricts the *audience* of a credential (or, more precisely, a message containing a knowledge formula). But to actually does more than that; a statement of the form

$$\text{Doris to Service}: \ [\varphi_1] \ \Leftarrow \varphi_2$$

has a non-declarative, operational meaning in DKAL. Essentially, it means that as soon as Doris manages to derive the condition φ_2 from *local* knowledge, it will send a message stating the formula φ_1 via a communication channel to Service, who will then import the formula as "A said φ_1" into local knowledge. Since the entire φ_1 is prefixed by "A said", the communicated message cannot depend on other principals' (and in particular, Service's) utterances. Therefore, protection against probing attacks in DKAL is achieved by means of restricting messages to those without delegated conditions; this is a rather harsh restriction.

The main motivation of DKAL2, then, is to remove DKAL's implicit restriction on messages [24]. Send assertions can now be of the form

$$\text{Doris to Service}: \ [\varphi_1 \ \leftarrow \ \varphi_1'] \ \Leftarrow \varphi_2$$

So when formula φ_2 is locally met, Doris sends the whole *conditional* message $\varphi_1 \ \leftarrow \ \varphi_1'$ to Service, where the condition φ_1' is interpreted as a condition on Service's local knowledge. But this opens the door to probing attacks. In an attempt to shut the door again, DKAL2 lets

Service define whitelist filters of the form

$$\text{Service from Doris}: \ [\psi_1 \ \leftarrow \ \psi_1'].$$

So Doris's conditional credential above is only imported into Service's local knowledge if $\varphi_1 \leftarrow \varphi_1'$ syntactically matches $\psi_1 \leftarrow \psi_1'$. This purely syntactic ad hoc matching mechanism is problematic as it breaks declarativeness (e.g., the message $\mathsf{ok} \leftarrow \mathbf{true}$ may cause a behaviour that is significantly different from the logically equivalent ok). Also, it is unreasonable to assume that Service can foresee all possible syntactic forms of incoming messages that may be acceptable. But, more importantly for the present discussion, it does not provide adequate protection against probing attacks, as the following example shows.

Suppose Service has the following send assertion in its policy:

$$\text{Service to } x : [\mathsf{ok}] \Leftarrow x \text{ can read}$$

In other words, Service will communicate the message "ok" to any principal x for which Service knows x can read.

Suppose Service wants to delegate authority over facts of the form x can read to Bob. Suppose further that Service wants to allow Bob to re-delegate this fact without restriction to other users. The users do not have constantly online communication channels with each other, so they need to express their re-delegation statements directly to Service.

Service then needs the following delegation assertion in its local knowledge store:

$$\text{Service} : (\text{Bob implied } x \text{ can read}) \rightarrow x \text{ can read}$$

This assertion means that if Service knows that Bob said x can read (perhaps indirectly, by derivation from other principals' utterances), then Service also knows x can read. Now suppose

we have the following re-delegation chain from Bob to Charlie and from Charlie to Doris:

Bob to Service : $[x^v$ can read $\leftarrow$ Charlie implied x^v can read$]$

Charlie to Service : $[x^v$ can read $\leftarrow$ Doris implied x^v can read$]$

In other words, Bob (and similarly, Charlie) communicates to Service that he is willing to say x can read if Service knows that Charlie (Doris, respectively) says so. (The v superscripts indicate that the variables should be left uninstantiated during the send operation.) But Service is only willing to accept such communications if suitable filter rules are in place. Filter rules are supposed to guard against probing attacks. But since Service does not know the re-delegation chain in advance, the most restrictive filters it can reasonably set up in this situation are these two:

Service from p : $[x$ can read $\leftarrow y$ implied x can read$]$

Service from p : $[x$ can read$]$

The first filter allows incoming re-delegation messages. The second filter is needed because the principal at the end of the chain may simply say the fact without any condition.

But now anyone in this re-delegation chain, for example Doris, can check whether anyone else, Y, say, has said that Doris can read to Service. The attack works as follows. Suppose Service has so far not communicated ok to Doris. Now Doris creates the following send assertion:

Doris to Service : [Doris can read $\leftarrow$ Y implied Doris can read]

If Service responds with ok (and Doris can link the response to her sending action), Doris can infer that Service knows that Y said Doris can read. In general, there may not be any trust relationship between Doris and Y, so this would constitute a breach of confidentiality. The same attack can be mounted by Bob and Charlie.

One possible reply is that the scenario is atypical for DKAL, because DKAL is not made for scenarios with connectivity restrictions between participants. (Recall that Bob, Charlie and Doris do not have constant open connections with each other and thus communicate directly

with Service.) But this is hardly reassuring, for it is still unclear which scenarios precisely are "typical" and thus safe, and what "safe" means in DKAL.

Moreover, the problem is not restricted to this kind of scenario. The attack is in fact possible in most situations where conditional messages are accepted. In most cases, the recipient of the message cannot foresee what the condition will be, and thus will be forced to write a very liberal, and thus dangerous, filter rule.

There is also another reason why DKAL's filter rules cannot give adequate protection against probing attacks, in comparison to the SecPAL solution given in Section 5. In the latter, it is the issuer of the credential who has the responsibility of controlling its dissemination using the "can listen to" operator. This makes sense because it is the issuer's confidentiality that is at stake here. In the DKAL solution, however, it is the responsibility of the receiver of the messages to make sure that the received information does not get imported by a condition within a message of an attacker. The sender has no way of restricting the subsequent information flow of his message within the receiver's derivations. He thus not only has to trust the receiver to run DKAL correctly (which is reasonable), but also has to trust the receiver to have filter rules in place that match his requirements (which is unreasonable).

3 Non-interference and Opacity

The goal of this section is to establish basic concepts for reasoning about information flow in declarative credential policies. We define a *policy language* $\mathcal{L}$ as a triple $(\mathcal{A}, \mathcal{Q}, \vdash)$ where $\mathcal{A}$ and $\mathcal{Q}$ are countable sets called *assertions* and *queries*, respectively, and the binary relation $\vdash \subseteq \wp(\mathcal{A}) \times \mathcal{Q}$ is the *decision relation*. Note that the decision relation need not be monotonic, although the concrete ones considered in later sections of this article are all monotonic. A *policy* A is a subset of $\mathcal{A}$.

At this stage, our definition of policy language is kept very abstract in order to cover a wide range of existing declarative policy languages. In later sections, we will concretise this concept and consider concrete languages or specific classes of languages.

For instance, in the language of Datalog (cf. Section 4), assertions are Horn clauses, queries

are boolean formulas over ground atoms, and $A \vdash q$ holds iff q holds in the unique minimal model of A. To model XACML [32], assertions would correspond to Rules, Policy Target declarations and references to Combining Algorithms; a query would model a Request together with a Response (Permit, Deny, Indeterminate, NotApplicable); and the decision relation would model the Policy Determination Point (PDP).

We fix a complete lattice of *security labels* $(\Lambda, \leq)$, and a *security environment* $\Gamma = (C, I)$, where $C : \mathcal{A} \cup \mathcal{Q} \to \Lambda$ is called the *confidentiality mapping*, and $I : \mathcal{A} \to \Lambda$ is the *integrity mapping*. We generalize C and I to sets by computing the least upper bound on the labels of the elements, e.g. $C(A) = \mathbf{lub}\{C(x) \mid x \in A\}$.

All definitions in this section are parameterized by $\mathcal{L}$, Λ and Γ; they are left implicit for the sake of readability. For the remainder of this article, let ℓ denote a label in Λ. Given a policy A, we define $\mathbf{visible}_\ell(A) = \{a \in A \mid C(a) \leq \ell\}$.

The security environment can be seen as defining a multi-level security meta-policy on the assertions within a service's local policy. The labels represent adversaries with varying clearance levels. The confidentiality mapping $C(a)$ specifies the minimal label, or the weakest adversary, capable of viewing (i.e., knowing) a specific assertion a inside a service's policy. The set $\mathbf{visible}_\ell(A)$ therefore denotes the part of A that can be passively observed (and reasoned about) by an adversary of level ℓ. In many systems, this set will be empty, or only contain the public access rules but not the extensional facts.

Additionally, an adversary may actively infer information about a policy by running a *probe*, that is, submitting a set of credentials (which are assertions) and evaluating a query against the *union* of the policy *and* the credentials. An adversary can gain more information by running a sequence of probes, but the set of probes available to the adversary is typically restricted: $C(q)$ is the minimal label needed to evaluate (and read the result of) the query q. For instance, in SecPAL [5], this part of C would model the Authorization Query Table, which exposes a list of queries to the public and prohibits evaluation of any other query.

The integrity mapping $I(a)$ denotes the minimal label required for being able to submit an assertion a as a credential. To model a typical setting for decentralized systems where assertions are statements "said" (or issued) by a principal, Λ could be defined as the powerset of the set

(a_1) Service says x can park if x consents to parking rules

(a_2) Service says x can park if x is a secret agent

(a_3) Service says Bob is a secret agent

(a_4) Service says Bob consents to parking rules

(q_1) Service says Bob can park

$C(x) = \text{Lo}$ if $x \in \{a_1, a_4, q_1\}$, and Hi otherwise.
$I(x) = \text{Lo}$ if $x = a_4$, and Hi otherwise.

Figure 1: Running example: the policy language is SecPAL [5], and Λ is the two-point lattice Lo $\leq$ Hi.

of principals, ordered by the superset ordering (so the set of all principals is bottom and the empty set is top). The integrity label $I(a)$ of a credential a issued and signed by a principal p would typically always include p (since p can freely create such an assertion), and additionally any other principal p' that is in possession of a (for instance, because p has issued and given credential a to p').

Definition 3.1 (Alikeness, observational equivalence)**.** Two policies A and A' are *alike up to* ℓ ($A \simeq_\ell A'$) iff $\mathbf{visible}_\ell(A) = \mathbf{visible}_\ell(A')$.

Two policies A and A' are *observationally equivalent up to* ℓ ($A \equiv_\ell A'$) iff

1. $A \simeq_\ell A'$, and
2. for all assertion sets $A'' \subseteq \mathcal{A}$ and queries $q \in \mathcal{Q}$ such that $I(A'') \leq \ell$ and $C(q) \leq \ell$:
   $A \cup A'' \vdash q \iff A' \cup A'' \vdash q$. □

A passive adversary of level ℓ can only see the ℓ-visible assertions, and hence cannot distinguish policies that are alike up to ℓ. An active adversary can see the visible assertions *and* run probes against the policy. These two capabilities are represented by conditions 1 and 2, respectively, in the definition of observational equivalence. Hence an active adversary of level ℓ cannot distinguish policies within any $(\equiv_\ell)$-equivalence class.

Example. Consider the assertions and the query from Fig. 1. Under the given security environment, $\{a_1\} \simeq_{\text{Lo}} \{a_1, a_2\} \simeq_{\text{Lo}} \{a_1, a_2, a_3\}$. As for observational equivalence, we have $\{a_1\} \equiv_{\text{Lo}} \{a_1, a_2\}$ because q_1 has the same outcomes in both policies, no matter whether the

Lo-integrity a_4 is injected or not. However, $\{a_1, a_2\} \not\equiv_{\text{Lo}} \{a_1, a_2, a_3\}$ because the latter satisfies q_1 (without injection of any assertion), whereas the former does not. □

Alikeness and observational equivalence are essential in the following definitions of two information flow properties, non-interference and opacity.

Definition 3.2 (Non-interference)**.** A policy A has the *non-interference property for* ℓ iff for all policies A':

$$A \simeq_\ell A' \Rightarrow A \equiv_\ell A'. \qquad \square$$

Informally, a policy has the non-interference property if any policy that looks the same to a passive adversary also behaves the same way when probed by an active adversary. This formulation based on observational equivalence is inspired by the definition of non-interference in transition systems by Zdancewic and Myers [41]. But the definition is also equivalent to the more informal way of describing non-interference: low output (i.e., the results of evaluating low queries) only depends on low input (the immutable visible part of the policy and the submitted assertions); in particular, it is independent of high input (i.e., the confidential assertions in the policy).

Example. Continuing the example above, neither $\{a_1\}$ nor $\{a_1, a_2\}$ nor $\{a_1, a_2, a_3\}$ has the non-interference property for Lo, but adding the Lo-visible a_4 to either of these policies results in a policy that has the non-interference property. This is because whenever a policy contains both a_1 and a_4, the only Lo-query q_1 is always true (since SecPAL is monotonic). □

Non-interference is a very restrictive property that, in the case of programming languages, rules out many innocuous programs that intentionally declassify confidential information. In our context, there are many useful policies in which the result of a low query intentionally depends on the presence or absence of some confidential information. For example, a patient may read an item from his medical record if it does not contain any confidential information about a third party. The result of checking if the patient's read request is permitted thus legitimately depends on a confidential fact, even if this fact should not be directly disclosed.

Another problem with non-interference is that it is too coarse. The policy mentioned above breaks non-interference, but all this tells us is that there is some dependency between the confidential information and the query result. It does not tell us what exactly the patient learns from the result. Indeed, he might not learn anything substantial from the fact that his read request was denied, because there may be several potential reasons for access denial, some of which may be unknown to him. Conversely, we could try to restore non-interference by classifying the confidential third party fact down to the patient's security label. This measure, however, would be too drastic, as it would fully disclose the third party information to the patient even without him probing.

What is needed is a property that provides a more fine-grained handle on what an active adversary may infer from a series of probes. This leads us to the information flow property of *opacity*, and its complement, which we call *detectability*. Opacity has been studied in the context of Petri nets and state transition systems [35, 11], where it is described as "the inability of an observer to establish the truth of a predicate over system traces". We reformulate this concept by replacing "system traces" by "policies".

Definition 3.3 (Detectability, opacity)**.** A *policy property* Φ is a set of policies. We often identify Φ with its indicator function, i.e., $\Phi(A)$ is equivalent to $A \in \Phi$.

A policy property Φ is ℓ-*detectable* in a policy A iff for all policies A':

$$A \equiv_\ell A' \Rightarrow \Phi(A').$$

A policy property Φ is ℓ-*opaque* in a policy A iff it is not ℓ-detectable in A, or equivalently, iff there exists a policy A' such that $A \equiv_\ell A'$ and $\neg\Phi(A')$. □

Intuitively, a policy property is ℓ-detectable in A if the adversary ℓ can infer (from the visible part of A and from running his probes against A) that A must have that property. Conversely, a policy property that holds in A is ℓ-opaque in A if the adversary cannot be sure that A has that property, because there exists some policy A' in which the property does not hold and that masquerades as A with respect to all probes available to ℓ.

Note that detectability requires knowledge with absolute certainty, and not simply high

probability. It may for instance be the case that "Alice is a secret agent or Bob is a nurse" is the smallest (i.e., most accurate) property that is detectable in the policy. From this it follows that the stricter property "Alice is a secret agent" is deemed opaque, simply because there is a possibility that Alice is not a secret agent but Bob is a nurse. In other words, opacity is based on a notion of uncertainty that is *possibilistic* as opposed to probabilistic.

Example. Under the given security environment from Fig. 1, the property that, according to the policy, Service says that Bob is a secret agent is Lo-opaque in $\{a_1, a_2, a_3\}$, and the property that Service does *not* say Bob is a secret agent is Lo-opaque in $\{a_1, a_2\}$. The property that Service does *not* say that Bob consents to parking rules is Lo-detectable in a_1 (by first injecting nothing, and then a_4), but Lo-opaque in $\{a_1, a_2, a_3\}$. □

The following is a list of basic sanity properties.

Proposition 3.4. Let $\ell, \ell' \in \Lambda$ with $\ell \leq \ell'$. The following propositions hold:

1. $A \simeq_{\ell'} A' \Rightarrow A \simeq_{\ell} A'$.
2. $A \equiv_{\ell'} A' \Rightarrow A \equiv_{\ell} A'$.
3. If A has the non-interference property for ℓ', then A also has it for ℓ.
4. If Φ is ℓ'-opaque in A, then Φ is also ℓ-opaque in A.
5. If Φ is ℓ-detectable in A, then Φ is also ℓ'-detectable in A.
6. If $\neg\Phi(A)$, then Φ is ℓ-opaque in A.
7. If Φ is ℓ-detectable in A, then $\Phi(A)$ holds.
8. If $\Phi' \subseteq \Phi$ and Φ is ℓ-opaque in A, then Φ' is ℓ-opaque in A.
9. If $\Phi' \supseteq \Phi$ and Φ is ℓ-detectable in A, then Φ' is ℓ-detectable in A.

Proof. These all follow directly from the definitions of alikeness, observational equivalence, opacity and detectability. □

What is the relationship between non-interference and opacity? Ryan and Peacock [35] showed, in the context of transition systems, that *non-inference* [33] can be cast in terms of opacity, but conjectured that the same cannot be easily done for non-interference. Theorem 3.6 below shows that non-interference can be precisely characterized in terms of opacity.

Definition 3.5 (Discriminating property)**.** Let A be a policy. A policy property Φ is (A, ℓ)-*discriminating* iff there exists a policy $A' \simeq_\ell A$ such that $\Phi(A) \iff \neg\Phi(A')$.

Theorem 3.6. A policy A has the non-interference property for ℓ iff all (A, ℓ)-discriminating policy properties are ℓ-opaque in A.

Proof. First, assume A has the non-interference property, and consider any (A, ℓ)-discriminating policy property Φ. If $\neg\Phi(A)$, then Φ is opaque in A anyway. In the other case, $\Phi(A)$, and by Def. 3.5, there exist $A' \simeq_\ell A$ such that $\neg\Phi(A')$. By non-interference, $A \equiv_\ell A'$. Hence Φ is ℓ-opaque in A.

For the other direction, assume the right hand side of the proposition, and consider any $A' \simeq_\ell A$. For the sake of contradiction, suppose $A \equiv_\ell A'$ does not hold. Then $\Phi = \wp(\mathcal{A}) \setminus \{A'\}$ is a (A, ℓ)-discriminating policy property. By opacity, there exists $A'' \equiv_\ell A$ such that $\neg\Phi(A'')$. By construction of Φ, $A'' = A'$, hence $A \equiv_\ell A'$, which contradicts the assumption. □

We have now defined two information flow properties for credential policies, non-interference and opacity. There is a crude and simple way of deriving a non-interference-enforcing decision relation $\vdash_\ell$ from the original decision relation $\vdash$: when evaluating a probe for ℓ, ignore all assertions in the policy that are higher than ℓ. More formally, for all policies A: $A \vdash_\ell q$ iff $\mathbf{visible}_\ell(A) \vdash q$. As a result, all A have the non-interference property for ℓ under $\vdash_\ell$. This method is safe if the language is monotonic in the sense that fewer assertions lead to fewer permissions.

Opacity (or likewise, detectability), on the other hand, is much harder to check, but also more flexible and suitable for our purpose. In fact, it is easy to see that opacity is generally undecidable for any policy language, as opacity is defined with respect to any arbitrary policy property, which may itself be undecidable.

4 Inferring Detectability

The scenario in Section 1 is an example of a probing attack through which an adversary can gain knowledge of confidential parts of a service's policy. The tools developed in the last section allow us to formally define and reason about this class of attacks. To do so effectively, the concepts of policy language and policy properties have to be concretized somewhat, while keeping them still abstract enough to be applicable to many concrete existing policy languages.

In this section, we will instantiate the policy language to Datalog [38, 12]. Datalog has a small syntax and semantics and can thus be reasoned about without much overhead, yet it is expressive enough to express a wide range of policies. It is the underlying semantics (though sometimes extended with constraints) for a number of policy languages [29, 27, 31, 16, 30, 8], and a number of others such as SecPAL [8] or fragments of DKAL [23] can be translated into it.

The concepts of opacity and detectability allow us to precisely quantify the information obtained through a probing attack. The last section ended on the remark that the generality of policy properties made opacity and detectability hard to check. In this section, we will therefore restrict policy properties to those linked directly to Datalog queries about the Datalog policy. Such properties include, for instance, "Bob is a secret agent *or* Mary does not have read access", but not "the policy has an odd number of assertions".

The main technical result of this section is an inference system for detectability of policy properties in Datalog policies. As was hinted at in Section 1, and which will become more evident in the following, analysing detectability is highly non-trivial, and it is easy to make mistakes. Given a set of probes, our inference system allows us to deduce what information about the policy is leaked. To be more precise, Theorem 4.8 is a soundness result in that it allows us to statically prove that an adversary can detect some policy property.

For safety analysis, it would be useful also to have the completeness result in order to prove opacity, i.e., that no adversary can detect some secret property. Whether the inference system for Datalog is complete or not remains an open problem (see discussion in Section 7), but Section 5 presents an alternative, practical solution for enforcing certain opacity guarantees.

4.1 Datalog

We fix a function-less first order signature with predicate names and constants. An *expression* e is a variable or a constant. An *atom* (or *fact*) $f = p(\vec{e})$ is a predicate name p applied to an expression tuple $\vec{e}$ of the correct arity. A *clause* is of the form

$$f_0 \leftarrow f_1, ..., f_n,$$

where $n \geq 0$, and f_0 is called the *head* and $\vec{f_i}$ the *body*. The arrow is usually omitted if $n = 0$. If a is a clause, then we write $\mathbf{hd}(a)$ to denote its head. A *program* is a set of clauses. A *query* q is either **true**, **false** or a ground (i.e., variable-free) boolean formula (i.e., involving connectives $\neg$, $\wedge$ and $\vee$) over atoms.

For the remainder of this section, we instantiate the set of assertions $\mathcal{A}$ to the set of Datalog clauses, and the set of queries $\mathcal{Q}$ to the set of Datalog queries. A policy is therefore a Datalog program. There are several equivalent definitions of the decision relation $\vdash$. The model-theoretic formulation is quite intuitive: given a policy $A \subseteq \mathcal{A}$, close each clause by universal quantifers and form the conjunction of all such quantified clauses. This formula has a unique minimal Herbrand model M. Then we define $A \vdash q$ iff $M \models q$.

For our proofs, a more operational definition is useful that is based on the intuition that Datalog programs are inductive definitions. Given a policy A, we define the *consequence operator* $\mathbf{T}_A$ as a monotonic mapping between sets S of ground atoms. In the following definition[1], let γ be a ground substitution (a total mapping from variables to constants).

$$\begin{aligned}\mathbf{T}_A(S) = \{\ f'_0\ \mid\ & \exists\gamma\ \exists\langle f_0 \leftarrow f_1, ..., f_n\rangle \in A,\\ & \gamma(\{f_1, ..., f_n\}) \subseteq S,\\ & f'_0 = \gamma(f_0)\ \}\end{aligned}$$

The least fixed point of the consequence operator, $\mathbf{T}^{\omega}_A(\emptyset)$, is equal to the model M above, so we can equivalently define $A \vdash q$ iff $\mathbf{T}^{\omega}_A(\emptyset) \models q$.

Datalog lends itself well to specifying policies because many policy rules can be naturally

[1] Throughout the article, we will occasionally use pointed brackets $\langle\text{-}\rangle$ as meta-level parentheses to delimit phrases of concrete syntax.

expressed as if-statements. For example, the SecPAL delegation assertion

$$\text{A says B can say } x \text{ has age } n \text{ if } x \text{ is a User}$$

could be translated[2] into two Datalog assertions, adopting the convention that the first parameter of each predicate specifies who "says" it:

$$\mathsf{canSayHasAge}(\mathrm{A}, \mathrm{B}, x, n) \leftarrow \mathsf{isA}(\mathrm{A}, x, \mathrm{User})$$

$$\mathsf{hasAge}(\mathrm{A}, x, n) \leftarrow \mathsf{hasAge}(\mathrm{B}, x, n), \mathsf{canSayHasAge}(\mathrm{A}, \mathrm{B}, x, n)$$

4.2 Probing environments

In Section 3, the adversary was characterized by a security label ℓ and a security environment $\Gamma = (C, I)$. The confidentiality mapping C defined which assertions were visible to her and which queries she could run, and the integrity mapping I defined which assertions she could submit and inject into the policy. But even though the adversary may have a large or infinite number of probes at her disposal, she will typically only use a subset of them in a particular probing attack. In this section, we will specify the adversary in a more fine-grained fashion as a specific set of probes, corresponding to a concrete probing attack.

Definition 4.1 (Probe, probing environment)**.** A *probe* $\pi = (A, q)$ is a pair consisting of an assertion set $A \subseteq \mathcal{A}$ and a query $q \in \mathcal{Q}$.

A *probing environment* $\Pi = (C, \ell, A_0, P)$ is a 4-tuple consisting of a confidentiality mapping $C : \mathcal{A} \to \Lambda$, a security label $\ell \in \Lambda$, a policy $A_0 \subseteq \mathcal{A}$, and a set P of probes. □

A probing environment (C, ℓ, A_0, P) is a complete specification of a probing attack. A_0 is the service's policy that is under attack; as before, C determines which assertions in A_0 are visible to the adversary, who is assumed to have label ℓ; and P is the set of probes that are run

[2]The actual translation according to [5] is actually a bit more complicated due to the fact that SecPAL supports a second kind of delegation construct, $\mathsf{can\ say}_0$, that forbids redelegation.

against A_0. The probe set P could be seen as a fine-grained instantiation of both the part of C that defined the queries available to the adversary and the integrity mapping I from Section 3.

We say that a probe (A, q) is *positive* if $A_0 \cup A \vdash q$, and *negative* otherwise. For each probe π in P, the adversary can observe if π is positive or negative. Based on the notion of probing environment, we generalize the definition of detectability.

Definition 4.2 (Observational equivalence)**.** Two policies A, A' are *observationally equivalent* under a probing environment $\Pi = (C, \ell, A_0, P)$ (we write $A \equiv_\Pi A'$) iff

1. $A \simeq_\ell A'$, and
2. for all probes (A'', q) in P:
   $A \cup A'' \vdash q \iff A' \cup A'' \vdash q$. □

Definition 4.3 (Detectability)**.** Let $\Pi = (C, \ell, A_0, P)$. A query q is Π-*detectable* iff for all policies A_0':

$$A_0' \equiv_\Pi A_0 \Rightarrow A_0' \vdash q.$$

A query q is Π-*opaque* iff it is not Π-detectable. □

This refined definition of detectability allows us to state succinctly what information about a policy is leaked to the adversary in a given probing attack.

4.3 An informal example

We now develop an inference system that proves if a query is detectable in a probing environment. To give an intuition for the problem, we first consider an example that is small and yet illustrates the surprising complexity of this problem. Suppose that the adversary cannot read anything from the policy A_0, so $\mathbf{visible}_\ell(A_0) = \emptyset$. Suppose further that the probe set consists of three probes that all have the same query, (A_1, ok), (A_2, ok), and (A_3, ok), where

$A_1 = \{\langle \mathsf{a} \rangle\}$, $A_2 = \{\langle \mathsf{b} \rangle\}$, and
$A_3 = \{\langle \mathsf{a} \leftarrow \mathsf{c} \rangle, \langle \mathsf{b} \leftarrow \mathsf{a} \rangle\}$.

(The nullary predicates a, b, c and ok in this example could represent any security-relevant facts in the policy.)

Now suppose that the first two probes are negative and the third is positive, i.e.,

$$A_0 \cup A_1 \nvdash \mathsf{ok}, \text{ and } A_0 \cup A_2 \nvdash \mathsf{ok}, \tag{1}$$

and

$$A_0 \cup A_3 \vdash \mathsf{ok}.$$

From these three probes alone, what can the adversary infer with certainty about the policy A_0?

First of all, what does the fact $A_0 \cup A_3 \vdash \mathsf{ok}$ convey? The probe would be trivially positive if ok were already true in A_0. But from (1) we can infer that actually $\neg\mathsf{ok}$ holds in $A_0 \cup A_1$, and thus also in A_0, by anti-monotonicity. So it must be the case that at least one of the two assertions in A_3 play an essential role in causing the third probe to be positive. Either the first assertion "fired", in which case a could not have already been true in A_0, and furthermore, c must hold. Or else the second assertion fired, in which case b is false in A_0, and either a or c are true, for these are the only two possibilities for firing the second assertion (we say: a and c form the *support of* b *in* A_3). In summary, $A_0 \cup A_3 \vdash \mathsf{ok}$ (together with (1)) implies

$$A_0 \vdash (\neg\mathsf{a} \wedge \mathsf{c}) \vee (\neg\mathsf{b} \wedge (\mathsf{a} \vee \mathsf{c})). \tag{2}$$

But there is more to be inferred. We can widen the two assertions in A_3 to $A_3' = \{\langle\mathsf{a}\rangle, \langle\mathsf{b}\rangle\}$ (we say: A_3 *is contained in* A_3') and, by monotonicity, be certain that

$$A_0 \cup A_3' \vdash \mathsf{ok}. \tag{3}$$

Note that A_1 and A_2 are proper subsets of A_3'. This fact can be exploited: from (3), we can

infer (again by arguing that either ok already holds or that the other assertion has fired):

$$\begin{array}{l} A_0 \cup \{\langle \mathsf{a} \rangle\} \vdash \mathsf{ok} \vee \neg \mathsf{b}, \text{ and symmetrically,} \\ A_0 \cup \{\langle \mathsf{b} \rangle\} \vdash \mathsf{ok} \vee \neg \mathsf{a}. \end{array} \tag{4}$$

Combining these with the two results from (1), we get $A_0 \cup \{\langle \mathsf{a} \rangle\} \vdash \neg \mathsf{b}$, and $A_0 \cup \{\langle \mathsf{b} \rangle\} \vdash \neg \mathsf{a}$. Then, by anti-monotonicity of the last two queries, we can infer $A_0 \vdash \neg \mathsf{a} \wedge \neg \mathsf{b}$. Finally, this can be combined with (2) and our knowledge of $\neg \mathsf{ok}$ to yield (after some algebraic simplification)

$$A_0 \vdash \neg \mathsf{ok} \wedge \neg \mathsf{a} \wedge \neg \mathsf{b} \wedge \mathsf{c}.$$

The adversary can thus infer that ok, a and b are false, and c is true in A_0.

4.4 The inference system

We now formalize the intuition given in the example above. In particular, we first need to make the notions of *monotonicity*, *containment*, *support*, and *firing* precise.

Definition 4.4 (Monotonicity, containment)**.** A query is *monotonic* iff it is equivalent to one without negation. A policy A is *contained in* a policy A' (we write: $A \preceq A'$) iff for all sets S of ground atoms, $\mathbf{T}^{\omega}_{A \cup S}(\emptyset) \subseteq \mathbf{T}^{\omega}_{A' \cup S}(\emptyset)$. □

A useful proposition that follows from this definition is that if q is monotonic and $A \preceq A'$ then for all policies A_0, $A \cup A_0 \vdash q$ implies $A' \cup A_0 \vdash q$. Conversely, if $A' \cup A_0 \vdash \neg q$ holds, then so does $A \cup A_0 \vdash \neg q$.

Containment is an undecidable problem in unrestricted Datalog. However, we will only check containment on ground policies, for which the problem is decidable (this follows from decidability results on containment in monadic Datalog [14]). Moreover, any conservative approximation of containment can be used without affecting soundness of the inference system (such as the syntactic widening hinted at in the example, or simply the subset relation).

Definition 4.5 (Support)**.** Let **support** be a function from any assertion set A and any ground atom f to a set of sets Δ of ground atoms such that the following holds:

1. If $\Delta \in \mathbf{support}(A, f)$ then $A \cup \Delta \vdash f$.

2. If $A \cup \Delta \vdash f$, then there exists $\Delta' \subseteq \Delta$ such that $\Delta' \in \mathbf{support}(A, f)$.

3. If $\Delta, \Delta' \in \mathbf{support}(A, f)$ and $\Delta' \subseteq \Delta$ then $\Delta = \Delta'$. □

The first two requirements represent soundness and completeness of **support**, respectively, and the third is minimality. This function can be computed directly using standard abduction [28]. Abduction has been used in AI applications such as planning and fault diagnosis, and has also been applied to security policy analysis for explaining access denial, for policy debugging [6] and for distributed credential gathering [2]. For unrestricted Datalog, $\mathbf{support}(A, f)$ may be infinite, which would result in infinite queries in our inference system. However, if A is finite and ground, then for all ground atoms f, $\mathbf{support}(A, f)$ is a finite set and all members of the set are finite sets of ground atoms. For this reason, we will restrict submitted assertions in probes to be ground, to prevent the inference system from generating judgements of infinite size.

Suppose A_1 is a partially unknown policy and A_2 a known ground policy. If we know that $A_1 \cup A_2 \vdash q$, what does this tell us about A_1? Either q already holds in A_1, or at least one of the assertions in A_2 is essential in proving q: it is "fired" in the context of A_1. More precisely, if f is the head of the fired assertion, then $\mathbf{fired}(A_2, f)$ holds in A_1 (Def. 4.6). Lemma 4.7 formalizes this argument, and is the cornerstone of the correctness proof for the inference system.

Definition 4.6 (Firing)**.** Let A be a ground assertion set, f a ground atom, and $S = \mathbf{support}(A, f)$. Then $\mathbf{fired}(A, f)$ denotes the query

$$\neg f \wedge \bigvee_{\Delta \in S} \bigwedge_{f_i \in \Delta} f_i. \qquad \square$$

Lemma 4.7. Let q be a query, and A_1, A_2 be policies where A_2 is ground. If $A_1 \cup A_2 \vdash q$ then $A_1 \vdash q \vee \bigvee_{a \in A_2} \mathbf{fired}(A_2, \mathbf{hd}(a))$.

Proof. Assume the left hand side of the proposition, and $A_1 \nvdash q$. It remains to show that the big disjunction is derivable in A_1. Let n be the smallest integer such that $\mathbf{T}^n_{A_1 \cup A_2}(\emptyset) \neq \mathbf{T}^n_{A_1}(\emptyset)$. Then by definition of the consequence operator, there must exist a ground assertion $\langle f \leftarrow$

$$(\textsc{Peek})\ \frac{\mathbf{visible}_\ell(A_0) \cup A \vdash q \quad q \text{ is monotonic} \quad A \text{ is ground}}{(C, \ell, A_0, P), A \Vdash q} \qquad (\textsc{Weak})\ \frac{\Pi, A \Vdash q \quad \models q \Rightarrow q'}{\Pi, A \Vdash q'}$$

$$(\textsc{Poke1})\ \frac{A_0 \cup A \vdash q \quad (A, q) \in P}{(C, \ell, A_0, P), A \Vdash q} \qquad (\textsc{Poke2})\ \frac{A_0 \cup A \nvdash q \quad (A, q) \in P}{(C, \ell, A_0, P), A \Vdash \neg q}$$

$$(\textsc{Mono1})\ \frac{\Pi, A \Vdash q \quad A' \succeq A \quad q \text{ is monotonic} \quad A' \text{ is ground}}{\Pi, A' \Vdash q} \qquad (\textsc{Mono2})\ \frac{\Pi, A \Vdash q \quad A' \preceq A \quad \neg q \text{ is monotonic} \quad A' \text{ is ground}}{\Pi, A' \Vdash q}$$

$$(\textsc{Conj})\ \frac{\Pi, A \Vdash q_1 \quad \Pi, A \Vdash q_2}{\Pi, A \Vdash q_1 \wedge q_2} \qquad (\textsc{Diff})\ \frac{\Pi, A_1 \cup A_2 \Vdash q}{\Pi, A_1 \Vdash q \vee \bigvee_{a \in A_2} \mathbf{fired}(A_2, \mathbf{hd}(a))}$$

Figure 2: Inference system for detectability.

$f_1, ..., f_m\rangle$ in A_2 such that $A_1 \vdash \neg f$ and all f_i are in $\mathbf{T}^{n-1}_{A_1 \cup A_2}(\emptyset) = \mathbf{T}^{n-1}_{A_1}(\emptyset)$ (the equality holds by construction of n). Moreover, by definition of **support**, there exists $\Delta \in \mathbf{support}(A_2, f)$ such that all f_i are in Δ. Hence the big disjunction holds. □

We now have all the tools needed to assemble the inference system for checking detectability (Fig. 2). The inference system generates judgements of the form $\Pi, A \Vdash q$ where $\Pi = (C, \ell, A_0, P)$ is a probing environment, A a set of ground assertions, and q a query. This judgement states that the adversary can learn that q holds in $A_0 \cup A$, just from looking at the visible part of the policy and from running the probes P. More precisely, the essential property of the inference system is as follows:

$$\begin{array}{l}\text{If } \Pi, A \Vdash q \text{ then for all policies } A'_0\text{:} \\ \qquad A'_0 \equiv_\Pi A_0 \;\Rightarrow\; A'_0 \cup A \vdash q.\end{array} \tag{5}$$

The axiom (Peek) models the knowledge gained from just passively reading the visible part of the policy. (Weak) allows the conclusion to be weakened; this rule is useful when the premise of another rule requires the query to be monotonic. The axioms (Poke1) and (Poke2) model the positive or negative result from a single probe from P. (Mono1) and (Mono2) exploit the properties of monotonicity and containment. (Conj) allow conclusions to be conjoined if they hold in the same credential context A. The last rule, (Diff), is the one that does most of the work: it encapsulates the implication of Lemma 4.7.

$$\dfrac{\dfrac{}{\Pi, A_3 \Vdash \mathsf{ok}}\,(\textsc{Poke1})}{\Pi, \emptyset \Vdash \mathsf{ok} \vee (\neg \mathsf{a} \wedge \mathsf{c}) \vee (\neg \mathsf{b} \wedge (\mathsf{a} \vee \mathsf{c}))}\,(\textsc{Diff})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Pi, A_3 \Vdash \mathsf{ok}}\,(\textsc{Poke1})}{\Pi, \{\langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle\} \Vdash \mathsf{ok}}\,(\textsc{Mono1})}{\Pi, \{\langle \mathsf{b} \rangle\} \Vdash \mathsf{ok} \vee \neg \mathsf{a}}\,(\textsc{Diff}) \qquad \dfrac{}{\Pi, \{\langle \mathsf{b} \rangle\} \Vdash \neg \mathsf{ok}}\,(\textsc{Poke2})}{\Pi, \{\langle \mathsf{b} \rangle\} \Vdash \neg \mathsf{a} \wedge \neg \mathsf{ok}}\,(\textsc{Conj})}{\Pi, \emptyset \Vdash \neg \mathsf{a} \wedge \neg \mathsf{ok}}\,(\textsc{Mono2})$$

Figure 3: Subproofs for the example from Section 4.3.

Finally, Theorem 4.8 formalizes the correctness of the inference system.

Theorem 4.8. Let q be a query and $\Pi = (C, \ell, A_0, P)$ be a probing environment where P is ground. If $\Pi, \emptyset \Vdash q$ then q is Π-detectable.

Proof. We prove the more general statement (5) above. The proof proceeds by rule induction on $\Vdash$. The interesting case is (Diff). Suppose $\Pi, A_1 \cup A_2 \Vdash q$. Consider any $A_0' \equiv_\Pi A_0$. By the induction hypothesis, $A_0' \cup A_1 \cup A_2 \vdash q$. Hence Lemma 4.7 can be applied to yield $A_0' \cup A_1 \vdash q \vee \bigvee_{a \in A_2} \mathbf{fired}(A_2, \mathbf{hd}(a))$. □

Example. Returning to the example from Section 4.3, we show how the proof of detectability proceeds according to the inference system. First of all, let $\Pi = (C, \ell, A_0, P)$ where $P = \{(A_1, \mathsf{ok}), (A_2, \mathsf{ok}), (A_3, \mathsf{ok})\}$, and A_0 is such that the first two probes are negative and the third one positive. Furthermore, C and ℓ are such that $\mathbf{visible}_\ell(A_0) = \emptyset$.

Fig. 3 shows two of the three essential subproofs. The third one, concluding in $\Pi, \emptyset \Vdash \neg \mathsf{b} \wedge \neg \mathsf{ok}$, is symmetric to the second subproof (swap a and b). The conjunction of all three conclusions yields $\neg \mathsf{ok} \wedge \neg \mathsf{a} \wedge \neg \mathsf{b} \wedge \mathsf{c}$, which, by Theorem 4.8, is Π-detectable.

4.5 Secret agents revisited

We now use the inference system to analyze the informal secret agent example from Section 1. We first translate the SecPAL probes into Datalog. The two assertions translate into the

credential set A_1 containing:

(1) $\mathsf{consent}(\mathrm{A}, \mathrm{A}) \leftarrow \mathsf{secret}(\mathrm{A}, \mathrm{B})$

(2) $\mathsf{secret}(\mathrm{A}, \mathrm{B}) \leftarrow \mathsf{secret}(\mathrm{S}, \mathrm{B})$

where A, B, S stand for Alice, Bob and Service, respectively, and the first parameter of each predicate represents the issuer of the fact. The query translates into

$q = \mathsf{canPark}(\mathrm{S}, \mathrm{A})$.

Suppose the first probe consists of both assertions (1) and (2), and the query q. Suppose that the probe is positive. From (Poke1), we establish the trivial fact that q succeeds with the probe.

Note that

$$\mathbf{support}(A_1, \mathsf{consent}(\mathrm{A}, \mathrm{A})) = \{\{\mathsf{secret}(\mathrm{A}, \mathrm{B})\}, \{\mathsf{secret}(\mathrm{S}, \mathrm{B})\}\}, \text{ and}$$

$$\mathbf{support}(A_1, \mathsf{secret}(\mathrm{A}, \mathrm{B})) = \{\{\mathsf{secret}(\mathrm{S}, \mathrm{B})\}\}.$$

Then from (Diff), we can infer that

$$q \vee (\neg\mathsf{consent}(\mathrm{A}, \mathrm{A}) \wedge (\mathsf{secret}(\mathrm{A}, \mathrm{B}) \vee \mathsf{secret}(\mathrm{S}, \mathrm{B}))) \vee (\neg\mathsf{secret}(\mathrm{A}, \mathrm{B}) \wedge \mathsf{secret}(\mathrm{S}, \mathrm{B}))$$

must be true in S's policy. There is nothing more to infer at this point. Contrary to the claim in [23], the adversary *cannot* infer $\mathsf{secret}(\mathrm{S}, \mathrm{B})$ from this probe alone. But suppose the adversary submits another probe, this time only with credential (1), and the result of the probe is negative.

Then we can use (Diff) on the first probe to obtain that

$$q \vee (\neg\mathsf{secret}(\mathrm{A}, \mathrm{B}) \wedge \mathsf{secret}(\mathrm{S}, \mathrm{B}))$$

holds true in S's policy *supplemented with* (1). With (Poke2) and (Conj) we can get rid of the

q, since we know that $\neg q$ is true in S's policy supplemented with (1). Hence the adversary can now infer with certainty that

$$\neg q \wedge \neg \mathsf{secret}(\mathrm{A}, \mathrm{B}) \wedge \mathsf{secret}(\mathrm{S}, \mathrm{B})$$

is true in S's policy. The last of these atoms corresponds to $\langle$S says B is a secret agent$\rangle$.

4.6 Detecting the absence of a fact

The secret agent example shows how delegation can be exploited to obtain confidential facts that hold in a policy. But how do we detect if a fact does *not* hold?

Suppose the first probe from the secret agent example above (consisting of assertions (1) and (2)) results in a *negative* response. The only fact that can be inferred from this negative probe is that q does not hold in S's policy (not even when supplemented with A_1). But the adversary could submit a second probe with just one credential:

$$A_2 = \{\mathsf{consent}(\mathrm{A}, \mathrm{A})\}.$$

If this probe succeeds, the adversary can be certain that Bob is *not* a secret agent. The following explains why.

We can use (Mono1) to deduce that q would also succeed with a credential set $A_3 = A_1 \cup \{\mathsf{secret}(\mathrm{S}, \mathrm{B})\}$, since $A_2 \preceq A_3$. Then applying (Diff) to this observation, we obtain that

$$q \ \vee \ \neg \mathsf{secret}(\mathrm{S}, \mathrm{B})$$

holds in S's policy in union with A_1. But we already know that $\neg q$ holds in this context, so with (Conj) we get

$$\neg \mathsf{secret}(\mathrm{S}, \mathrm{B}),$$

which also holds in the original policy due to (Mono2).

4.7 The weakening rule

The weakening rule (Weak) from Section 4 may not seem very useful at first sight. We now show an example of an inference that makes essential use of (Weak).

Suppose the policy under attack is A_0, and none of its assertions are visible to the adversary. The adversary runs two probes with credential sets

$A_1 = \{\langle \mathrm{a} \leftarrow \mathrm{b}\rangle, \langle \mathrm{c} \leftarrow \mathrm{a}\rangle\}$ and
$A_2 = \{\langle \mathrm{c} \leftarrow \mathrm{a}\rangle\}$,

respectively, and both probes use the query $\langle \mathsf{ok} \rangle$. Suppose the first probe is positive, and the second is negative.

First of all, applying (Mono2) on the second probe establishes $\neg\mathsf{ok}$ in A_0, so we can drop ok in the following deductions.

Applying (Diff) to the first probe, we get that

$$(\neg \mathrm{a} \wedge \mathrm{b}) \vee (\neg \mathrm{c} \wedge \mathrm{a})$$

is true in A_0. Again applying (Diff) to the first probe, we can also get that

$$\neg \mathrm{a} \wedge \mathrm{b}$$

is true in $A_0 \cup A_2$.

Now we use (Weak) on this result, which tells us that $\neg$a is true in $A_0 \cup A_2$. The purpose of applying (Weak) at this point is to project the query onto an anti-monotonic query, so that (Mono2) can now be applied to obtain that $\neg$a is also true in A_0. Finally, in conjunction with the earlier deduction, the adversary can infer with certainty that $\neg \mathrm{a} \wedge \mathrm{b}$ is true in A_0.

$$(\textsc{Cond})\ \frac{\langle e\ \mathsf{says}\ f\ \mathsf{if}\ f_1, ..., f_n\ \mathsf{where}\ c\rangle \in A \quad \models \gamma(c) \qquad A \vdash \gamma(e\ \mathsf{says}\ f_i)\ \text{ for all } i \in \{1...n\}}{A \vdash \gamma(e\ \mathsf{says}\ f)}$$

$$(\textsc{CanSay})\ \frac{A \vdash e_1\ \mathsf{says}\ e_2\ \mathsf{can\ say}\ f \qquad A \vdash e_2\ \mathsf{says}\ f}{A \vdash e_1\ \mathsf{says}\ f} \qquad (\textsc{CanSay0})\ \frac{A \vdash e_1\ \mathsf{says}\ e_2\ \mathsf{can\ say}_0\ f \qquad \{a \in A \mid e_2 \text{ is the issuer of } a\} \vdash e_2\ \mathsf{says}\ f}{A \vdash e_1\ \mathsf{says}\ f}$$

Figure 4: SecPAL proof system.

5 Information flow control

We now turn to the question as to how information flow can be controlled in a "real" policy language, namely SecPAL. SecPAL lends itself well to this purpose, as it has been designed with the goal of maximising generality and expressiveness while keeping the number of primitive constructs minimal. But more importantly, it is paradigmatic in its susceptibility to probing attacks based on delegation. This vulnerability is shared by almost the entire family of related languages that support the decentralized delegation of authority via the says-operator, first introduced in the ABLP logic [1], or any equivalent construct.

We briefly recall SecPAL, before taking a closer look at what causes the vulnerability. We then show how this vulnerability can be mitigated by a mild modification of the way delegation works in SecPAL. This method can be easily transferred to other policy languages that support delegation.

5.1 SecPAL

We give a very brief overview of SecPAL; for a more careful treatment, see [3] and [5].

Syntax. We fix an arbitrary first-order function-less signature with countably infinite sets of predicate names and constants (including principals). An *expression* e is either a variable or a constant. A *fact* f is either *flat*, i.e., it is a predicate atom (which we often write in infix notation, e.g. $e\ \mathsf{can\ read}\ e'$), or *nested*, i.e., it is of the form $e\ \mathsf{can\ say}\ f'$ or $e\ \mathsf{can\ say}_0\ f'$, where f' may be nested itself.

We further fix an arbitrary constraint language. A constraint c is a first-order formula over

atomic constraints (such as equality, inequalities, arithmetic constraints or regular expression constraints). The only requirement on the constraint language is that it be equipped with a computable unary relation $\models$, such that $\models c$ holds whenever the ground constraint c is true. SecPAL can thus be flexibly adapted to various specific domains by choosing a domain-specific predicate name set and constraint language.

An *assertion* a is of the form

$$e \text{ says } f \text{ if } f_1, ..., f_n \text{ where } c,$$

where $n \geq 0$, e is ground and the f_i are flat. We say that e is the *issuer* of this assertion. We omit the "if" if $n = 0$ and the "where" if $c = \mathbf{true}$. An assertion of the form $\langle e \text{ says } f \rangle$ is an *atomic* assertion.

A *query* is a boolean formula over ground atomic assertions involving only flat facts. (SecPAL actually allows non-ground, constrained and quantified queries, but we omit these for simplicity.)

For the remainder of this section, we instantiate $\mathcal{A}$ to the set of SecPAL assertions and $\mathcal{Q}$ to the set of SecPAL queries.

Decision relation. Fig. 4 shows the decision relation $\vdash$ for atomic queries (i.e., ground atomic assertions). In the modus ponens rule (Cond), γ is a ground substitution. Note that (Cond) requires the conditional facts f_i to be issued by the same principal e that issues the original assertion and the conclusion fact. The only way that foreign facts may enter the derivation is via (CanSay) and (CanSay0). The former implements standard delegation of authority: e_1 delegates authority over f to e_2, so e_1 is willing to vouch for f whenever e_2 says it. The latter is similar, but prohibits redelegation: the delegator e_2 must say f directly, without any dependencies on what other principals say.

If A is a policy, then let $[\![A]\!]$ denote the set of atomic queries a such that $A \vdash a$. Then for a general query q, we define $A \vdash q$ iff $[\![A]\!] \models q$. We refer the interested reader to [5] for a collection of example policies and policy idioms expressed in SecPAL.

5.2 Delegation-based probing attacks

From Section 4, it is clear that in all but the most trivial circumstances, any confidential fact in the policy can be detected with the right combination of probes, even if the query does not mention the fact at all. Consider for instance the following two SecPAL probes:

$\pi_1 = (\{\text{A says ok if secret}\}, \text{A says ok})$

$\pi_2 = (\emptyset, \text{A says ok})$

If π_1 is positive and π_2 is negative, then $\langle \text{A says secret} \rangle$ can be detected by any B running these probes. But since the essential credential in π_1 is issued by A herself, one could argue that the only one who could run this probe would be A herself, or perhaps A intended this information flow since she allowed this obviously dangerous credential to get into somebody else's possession (or else A is grossly careless).

But the real danger of probing lies in the fact that any confidential fact issued by A can be detected using probes that do not contain any assertions issued by A. Indeed, even the queries in the probes need not mention A at all, as the following example shows:

$\pi'_1 = (\{\text{B says ok if secret},$
$\quad \text{B says A can say secret}\}, \text{C says foo})$

$\pi'_2 = (\{\text{B says ok if secret}\}, \text{C says foo})$

If these assertions are translated into Datalog (as described in [5]), the inference system from Section 4 can be used to infer that $\langle \text{A says secret} \rangle$ is detectable if π'_1 is positive and π'_2 is negative.

Clearly, the underlying mechanism for this attack (and similar attacks) works by exploiting the ability to delegate authority over facts to others. In other words, these attacks depend on the rules (CanSay) and (CanSay0). Taking a closer look at the (CanSay) rule, note its original intention is for e_1 to specify who (e_2) can be trusted on saying f. The principal e_1 is willing to import f from e_2 if e_1 has issued the corresponding can say assertion. But the rule also reveals an apparent asymmetry between e_1 and e_2: e_1 has to explicitly agree to importing e_2's fact f, but there is no premise that requires e_2 to agree to exporting the fact to e_1. Hence e_1 can

import from e_2 without e_2's consent, under the guise of delegating authority over f to e_2.

5.3 SecPAL$^+$

Based on the intuition developed above, we now propose a mild modification to the two delegation rules which provides a more symmetric protection of both the delegator – the one who imports a fact – and the delegatee – the one who exports the fact. We first introduce a new type of nested fact of the form $\langle e \text{ can listen to } f \rangle$, where f is a (possibly nested) fact. The rules (CanSay) and (CanSay0) are then replaced by (CanSay$^+$) and (CanSay0$^+$) (below, A_{e_2} denotes $\{a \in A \mid e_2 \text{ is the issuer of } a\}$):

$$(\text{CanSay}^+)\ \frac{\begin{array}{c} A \vdash e_1 \text{ says } e_2 \text{ can say } f \\ A \vdash e_2 \text{ says } e_1 \text{ can listen to } f \\ A \vdash e_2 \text{ says } f \end{array}}{A \vdash e_1 \text{ says } f}$$

$$(\text{CanSay0}^+)\ \frac{\begin{array}{c} A \vdash e_1 \text{ says } e_2 \text{ can say}_0 f \\ A_{e_2} \vdash e_2 \text{ says } e_1 \text{ can listen to } f \\ A_{e_2} \vdash e_2 \text{ says } f \end{array}}{A \vdash e_1 \text{ says } f}$$

We call the language with the modified delegation rules SecPAL$^+$. Consider the following (re-)delegation chain:

A says B can say f

B says C can say f

C says D can say f

D says f

In standard SecPAL, these assertions would suffice to derive A says f. In SecPAL$^+$, we need additional assertions to make the delegation succeed:

B says A can listen to f

C says B can listen to f

D says C can listen to f

This conforms nicely with the idea of a delegation chain, in which each principal only knows the peer above and below the chain. The final delegatee D, for instance, need not specify that B and A can listen to f.

Opacity in SecPAL$^+$. The goal of modifying SecPAL's delegation mechanism was to rule out probing attacks based on delegation. But what exactly do our modifications achieve? We can answer this question in terms of opacity and detectability.

In the following, let Λ be the two-point lattice Lo $\leq$ Hi. For simplicity, we let C be a confidentiality mapping such that $C(a) = \text{Hi}$, for all assertions a. Hence for any policy A_0, $\mathbf{visible}_{\text{Lo}}(A_0) = \emptyset$; we thus assume that the entire policy is invisible to a low passive adversary.

Theorem 5.1 below provides a simple opacity guarantee for a confidential fact q_0 that is issued by some e_1. Assuming that (1) the adversary is not permitted to directly query q_0, (2) the adversary is not in possession of any obviously compromising assertion issued by e_1 (cf. Section 5.2), and (3) the adversary is not in possession of a $\langle$can listen to$\rangle$-assertion for q_0 issued by e_1, the theorem guarantees opacity of q_0.

Theorem 5.1. Let q_0 be a possibly negated atomic query of the form $(\neg)(e_0 \text{ says } p(\vec{e}))$, and $\Pi = (C, \text{Lo}, A_0, P)$ a (SecPAL$^+$) probing environment. If for all $(A, q) \in P$,

1. q_0 does not occur in q, and
2. $p(\vec{e})$ does not occur in any ground instance of an e_0-issued assertion $a \in A$, and
3. for all $a \in A$:

   $A_0 \cup A \nvdash e_0 \text{ says } e_a \text{ can listen to } p(\vec{e})$,

   where e_a is the issuer of a,

then q_0 is Π-opaque.

Proof. If $A_0 \nvdash q_0$, the statement is trivially true, so assume $A_0 \vdash q_0$. The main idea of the proof is to construct some A_0' such that (a) $A_0' \nvdash q_0$ and (b) $A_0' \equiv_\Pi A_0$.

First consider the case where q_0 is not negated. Let A_0' be obtained from A_0 by

- replacing all occurrences of p by some p', where p' does occurs neither in A_0 nor in P;

- adding assertions e says $p(\vec{x})$ if $p'(\vec{x})$, for all constants $e \neq e_0$ occurring in A_0 or P;
- adding the assertion e_0 says $p(\vec{x})$ if $p'(\vec{x})$ where $\vec{x} \neq \vec{e}$; and
- adding e says $p'(\vec{x})$ if $p(\vec{x})$, for all constants e occurring in A_0 or P.

Requirement (a) holds as there is no assertion in A_0' with a head that could produce the conclusion q_0.

To show that (b) holds, consider an arbitrary probe $(A, q) \in P$. Then, by a straightforward rule induction on $\vdash$, we have

$$\{q' \mid A_0' \cup A \vdash q'\} = Q_0 \cup Q_1 \setminus \{q_0\}, \tag{6}$$

where $Q_0 = \{q' \mid A_0 \cup A \vdash q'\}$ and $Q_1 = \{e \text{ says } p'(\vec{e}') \mid A_0 \cup A \vdash e \text{ says } p(\vec{e}')\}$. (The second assumption of the Theorem is used for the (Cond) case, and the third assumption for the cases (CanSay$^+$) and (CanSay0$^+$).) Now consider any atomic subexpression q'' of q. Suppose $A_0' \cup A \vdash q''$. Then by (6) and the fact that p' does not occur in q, we have $q'' \in Q_0$, and hence $A_0 \cup A \vdash q''$. For the other direction, suppose $A_0 \cup A \vdash q$. From the first assumption of the Theorem, $q'' \notin \{q_0\}$, hence by (6), $A_0' \cup A \vdash q''$, as required for (b).

In the other case where q_0 is negated, A_0' is constructed from A_0 by adding e_0 says $p(\vec{e})$ (and thus making (a) true) and by applying the additional following transformations (for making (b) true). For each e_0-issued assertion in which a fact of the form $p(\vec{e}')$ occurs in the body, replace the original constraint c by $c \wedge \vec{e} \neq \vec{e}'$. Furthermore, for each assertion with a head of the form $\langle e' \text{ can say } p(\vec{e}'')\rangle$, replace the original constraint c by $c \wedge e' \neq e_0 \wedge \vec{e}'' \neq \vec{e}$.

By rule induction on $\vdash$, we can then prove that for all $(A, q) \in P$,

$$\{q' \mid A_0' \cup A \vdash q'\} \setminus \{q_0\} = \{q' \mid A_0 \cup A \vdash q'\}. \tag{7}$$

Together with the first assumption of the Theorem, it then follows that $A_0 \cup A \vdash q$ iff $A_0' \cup A \vdash q$, which implies (b).

□

6 Related Work

Automated trust negotiation (ATN), first introduced by Winsborough et al. [40], is concerned with negotiation strategies for exchanging confidential credentials between mutual strangers. The notion of negotiation safety is formalized in Winsborough and Li [39], which also provides a comprehensive overview of research efforts in this area. Informally, a safe negotiation strategy does not reveal anything about the presence or absence of a credential (or an attribute) to the negotiation partner before the latter has proved that he satisfies the necessary disclosure conditions, specified by some policy. There are two main differences between the ATN setting and the current one which make the two hard to directly compare. Firstly, ATN considers the confidentiality of credentials of all involved parties, which necessitates a sequential and gradual negotiation process. In contrast, we are only concerned with the confidentiality of the service's policy. Secondly, work in ATN has so far not considered the possibility of agents *injecting* objects into the policy. Our adversaries, in contrast, can submit credentials that, in effect, logically extend the service's policy for the duration of one probe. This ability is natural in the context of trust management [9], but it also complicates the analysis.

Information flow has been studied since the mid 1970s [15], though research has mainly been concerned with stateful, temporal computations. A good survey is found in [36]. The current setting is quite different as there is no notion of state, state transition, run or trace, and adversaries are traditionally not permitted to inject code into the program. Nevertheless, on an abstract level, many of the traditional concepts can be adapted, such as non-interference (introduced by Goguen and Meseguer [21]). In fact, our definition is inspired by Zdancewic and Myers' definition [41], which is also based on observational equivalence (albeit on traces). Opacity [35, 11] is another, less known, information flow property that we reformulated and that proved very useful for our purposes; it is also closely related to non-inference [33] and non-deducibility [37].

The knowledge operator K_i in epistemic modal logic [18] bears some resemblance to our definition of detectability, and has been used for reasoning about secrecy in (stateful and temporally evolving) multi-agent systems [25]. In this logic, a formula Φ is known by agent i ($K_i\Phi$)

at state s iff in all states s' accessible by i from s (intuitively, these are all states that are considered possible by i), Φ holds. In our framework, the adversary gains partial knowledge about the policy by passively and actively observing a policy A. More precisely, she can narrow down the set of policies she considers as possible candidates to the set of all A' such that $A \equiv_\ell A'$. If Φ holds in all these "possible" policies, then she can effectively detect Φ in A.

The database inference problem is concerned with indirect inference channels through which confidential information from a database can leak to a database user. A wide variety of such channels have been identified and studied [26, 19]. In particular, Bonatti et al. have studied the inference problem in deductive databases [10], which are similar to declarative policies. However, they do not consider users who can temporarily inject new rules and relations into the database, as this is not natural in the database context.

The notions of probing attacks and detectability in Datalog policies (Section 4) are related to hypothetical logic programming [17], where goals (corresponding to our queries) may be supplemented by hypothetical clauses which are temporarily added to the logic program (corresponding to the submitted credentials). However, we are not just interested in evaluating such hypothetical goals (i.e., probes), but in finding an *explanation* for the result of these goals. Abduction, in its most general form, is about finding explanations to a set of observations [34], and has been extensively studied in the context of standard logic programming [28], but so far not of hypothetical reasoning. Indeed, the inference system in Section 4 could be viewed as an abduction method for hypothetical goals in Horn logic programs.

7 Discussion

In Section 3, we developed an information flow framework for credential-based policy systems, on the basis of alikeness (with respect to the visible part of a policy) and observational equivalence (with respect to probes). This abstract setting gave rise to very general and elegant formulations of the relevant information flow properties: a policy has the non-interference property if any policy that looks alike is observationally equivalent; and a policy property is detectable in a policy if any observationally equivalent policy has that property. We leave it to future work to

examine if some of the many other information flow properties [20] can also be usefully adopted.

Completeness. This framework was then put to good use in Section 4 where we formalized probing attacks and developed an inference system for checking detectability in Datalog policies. Theorem 4.8 proves the *soundness* of the system: whenever a query is derivable in the system, then it is also detectable. However, the corresponding *completeness* statement remains a conjecture:

Conjecture 7.1. Let q be a query and $\Pi = (C, \ell, A_0, P)$ be a probing environment where A_0 is finite and P is ground and finite. If q is Π-detectable then $\Pi, \emptyset \Vdash q$ holds.

Completeness would be a useful property because it would imply at least semidecidability of detectability (for ground Datalog probes), and thus detectability could be positively checked in finite time. Furthermore, if $\Pi, \emptyset \nVdash q$ holds, then q would also be known to be opaque in the policy. Of course, to check $\Pi, \emptyset \nVdash q$ in general would require full decidability, which is not automatically implied by completeness.

The reason why completeness is not easy to prove is that the premise of detectability, being a universally quantified implication, does not provide any obvious information that would guide the inference proof. In any case, the inference proofs do not seem to be very goal-directed, as the small example in Section 4.3 illustrated. In particular the combination of (Weak), (Mono1) and (Diff) contribute to the apparent unconstructiveness of the inference system.

A more promising strategy may be to prove the *modus tollens* direction: assume $\Pi, \emptyset \nVdash q$, and then construct a policy that masquerades as A_0 with respect to the probes, but in which q does not hold. While it is easy to modify A_0 in such a way that $\neg q$ holds, the difficulty here lies in the fact that it is not clear how to "repair" the modified policy in order to successfully masquerade as A_0.

Of course, it may turn out that completeness does not hold. However, a simple informal argument suggests that ground finite detectability is actually fully decidable: it seems almost certain that all maximally strong queries that are detectable only mention predicate names and constants that occur in the probes or the visible part of the policy. If this is true, then there are only finitely many maximally strong detectable queries, and all other detectable queries are

implications of these. Hence the set of detectable queries is decidable. So it seems that at least some complete finite axiomatization of detectability exists, although it may be an extension of the given inference system.

There are other interesting open problems apart from completeness that we leave for possible future work. In particular, we have not analyzed the complexity of the inference system, nor have we explored detectability in the context of a non-monotonic language such as Datalog with negation. Another limitation of the inference system is that it requires a rather precise specification of the adversary as input, namely a set of concrete credentials representing an upper bound on the credentials that are available to the adversary in a probing attack. A more powerful (but probably much harder) analysis would be to check detectability or opacity of a policy with respect to adversaries that are characterized by less concrete constraints on available credentials, e.g., adversaries who can create arbitrary self-issued credentials.

Alternative approaches to prevent attacks. In Section 5 we proposed a modification to SecPAL's delegation mechanism in order to control information leaks due to probing.

Cassandra [8, 7] is an earlier authorization language in which probing attacks are mitigated by design, albeit by much cruder means. In Cassandra, submitted credentials must not have any conditional facts. Furthermore, as in most other languages, delegation can only be expressed using conditional facts. Therefore, the only information that may be leaked by probing is the result of the query, and possibly that the submitted credential does not exists in the local policy as an assertion. The latter leakage is also possible in SecPAL$^+$: suppose the adversary submits no credentials and gets a negative result for a query, and subsequently submits a single credential a that does not contain any conditional facts, and receives a positive result for the same query. Then $\neg a$ is detectable in the policy. However, this scenario is usually not problematic, firstly, as the local policy must have been written explicitly to be dependent on a, and secondly, as the adversary's possession of a usually implies that a is not confidential to the adversary. But while Cassandra provides opacity guarantees similar to SecPAL$^+$, it does so at the cost of severely restricting the expressiveness of submitted credentials.

DKAL [23] uses a similar approach of restricting messages to non-conditional ones in order to prevent probing attacks, and at a similarly high cost. As we demonstrated in Section 2,

DKAL2 [24], which introduces conditional messages into the language, is actually vulnerable to the probing attack, because the mechanism of whitelist message filters is not strong enough to provide any meaningful opacity guarantees.

One of the features that sets DKAL apart from other languages is that its utterances are directed towards a specified audience. We initially explored if a similar mechanism could be used in SecPAL to prevent malicious probing attacks. Indeed, we found that this is possible without needing to commit to DKAL's operational, non-declarative semantics for "to". One possible solution in this direction would be to replace "says" by a directional "saysto" keyword, and the (CanSay) rule by the following rule (and similarly for (CanSay0)):

$$(\textsc{CanSay}') \frac{A \vdash e_1 \text{ saysto } e_3 : \ e_2 \text{ can say } f \qquad A \vdash e_2 \text{ saysto } e_1 : \ f}{A \vdash e_1 \text{ saysto } e_3 : \ f}$$

This approach was eventually abandoned because the SecPAL$^+$ approach scores higher in several dimensions of usability. In the saysto solution, policies are harder to change, as each single assertion is tagged with a specific audience, whereas in SecPAL$^+$, the can listen to assertions can be seen as a meta-policy on matching classes of facts. One may naively believe that the more a policy language allows users to finely control the meaning of their policies, the better it is. But as with most other aspects in designing a policy language, the goal is to get the balance right. If the level of semantic differentiation is too fine and subtle, policy authors will choose their parameters randomly or by gut feeling, will probably not understand their own policies in detail, and will likely write policies with unintended consequences. In the alternative solution with saysto, policy authors will likely be overwhelmed if forced to choose a saysto parameter for each and every assertion they write.

Furthermore, at the time when facts are added to the database or assertions are written, it is usually neither clear nor relevant who the audience should be. So the decision on the audience is forced at the wrong time, and at a wrong level of abstraction. Rather than specifying the audience on an assertion-by-assertion level, it is much simpler to view it as an orthogonal problem, specified on the coarser fact level.

Another problem is that there are many subtly different variants of (CanSay$'$), all of which are more or less plausible. This contingency makes it hard to understand and to write poli-

cies. In summary, the solution using directed utterances is problematic with respect to several usability dimensions, in particular viscosity, premature commitment, error-proneness, hard mental operations, and abstraction. (These terms are from the cognitive dimensions of notations framework [22].)

Conclusion. The case study in Section 2 illustrates what can go wrong if the fundamental security questions are not systematically answered [10]: what do we protect, against whom do we protect, what does "protect" mean, and how do we protect?

In this article, we provided a formal framework as a first step towards answering these questions. The design of the framework was driven by concepts from traditional information flow research. *What we protect* is expressed in terms of multi-level security labels on policy assertions and queries. *Against whom we protect* is specified by a formal definition of probe as a query together with a set of supporting credentials that are temporarily injected into the policy, and adversaries that differ in which probes are available to them, again induced by the security labels. We quantify the effective power of a specific adversary by means of an inference system that tells us what the adversary can detect about the policy. *What protection means* is stated in terms of non-interference, which we found to be too restrictive and coarse, and opacity, both based on a notion of observational equivalence. Finally, we provided a simple and elegant *protection method* that works by restoring symmetry in delegation of authority and comes with strong opacity guarantees.

Acknowledgements. I thank Cédric Fournet, Arne Heizmann, Andy Gordon and the anonymous referees for helpful comments and suggestions.

References

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, 1993.

[2] M. Becker, J. Mackay, and B. Dillaway. Abductive authorization credential gathering. In *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2009)*, 2009.

[3] M. Y. Becker. SecPAL formalisation and extensions. Technical Report MSR-TR-2009-127, Microsoft Research, 2009.

[4] M. Y. Becker. Information flow in credential systems. In *IEEE Computer Security Foundations Symposium*, pages 171–185, 2010.

[5] M. Y. Becker, C. Fournet, and A. D. Gordon. Design and semantics of a decentralized authorization language. In *IEEE Computer Security Foundations Symposium*, pages 3–15, 2007.

[6] M. Y. Becker and S. Nanz. The role of abduction in declarative authorization policies. In *10th International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2008.

[7] M. Y. Becker and P. Sewell. Cassandra: distributed access control policies with tunable expressiveness. In *IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 159–168, 2004.

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

[9] M. Blaze, J. Feigenbaum, and A. D. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming*, pages 185–210, 1999.

[10] P. Bonatti, S. Kraus, and V. Subrahmanian. Foundations of secure deductive databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):406–422, 1995.

[11] J. Bryans, M. Koutny, L. Mazaré, and P. Ryan. Opacity generalised to transition systems. *International Journal of Information Security*, 7(6):421–435, 2008.

[12] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[13] 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):285–322, 2001.

[14] S. Cosmadakis, H. Gaifman, P. Kanellakis, and M. Vardi. Decidable optimization problems for database logic programs. In *ACM Symposium on Theory of Computing*, pages 477–490, 1988.

[15] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5), 1976.

[16] J. Detreville. Binder, a logic-based security language. In *IEEE Symposium on Security and Privacy*, pages 105–113, 2002.

[17] P. Dung. Declarative semantics of hypothetical logic programming with negation as failure. *Lecture Notes in Computer Science*, pages 45–58, 1993.

[18] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. Reasoning About Knowledge. 2003.

[19] C. Farkas and S. Jajodia. The inference problem: a survey. *ACM SIGKDD Explorations Newsletter*, 4(2):6–11, 2002.

[20] R. Focardi and R. Gorrieri. A classification of security properties. *Journal of Computer Security*, 3(1):5–33, 1995.

[21] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and privacy*, volume 12, 1982.

[22] T. Green. Cognitive dimensions of notations. In *People and Computers V: Proceedings of the Fifth Conference of the British Computer Society Human-Computer Interaction Specialist Group*, page 443. Cambridge University Press, 1989.

[23] Y. Gurevich and I. Neeman. DKAL: Distributed-knowledge authorization language. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 149–162, 2008.

[24] Y. Gurevich and I. Neeman. DKAL 2 – a simplified and improved authorization language. Technical Report MSR-TR-2009-11, Microsoft Research, 2009.

[25] J. Halpern and K. O'Neill. Secrecy in multiagent systems. *ACM Transactions on Information and System Security (TISSEC)*, 12(1), 2008.

[26] S. Jajodia and C. Meadows. Inference problems in multilevel secure database management systems. *Information Security: An Integrated Collection of Essays*, pages 570–584, 1995.

[27] T. Jim. SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 106–115, 2001.

[28] A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235–324, 1998.

[29] N. Li, B. Grosof, and J. Feigenbaum. A practically implementable and tractable delegation logic. In *IEEE Symposium on Security and Privacy*, pages 27–42, 2000.

[30] N. Li and J. C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Practical Aspects of Declarative Languages*, pages 58–73, 2003.

[31] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust management framework. In *Symposium on Security and Privacy*, pages 114–130, 2002.

[32] OASIS. *eXtensible Access Control Markup Language (XACML) Version 2.0 core specification*, 2005.

[33] C. O'Halloran. A calculus of information flow. In *Proceedings of the European Symposium on Research in Computer Security, Toulouse, France*, 1990.

[34] C. S. Peirce. Abduction and induction. In J. Buchler, editor, *Philosophical Writings of Peirce*. Dover Publications, Oxford, 1955.

[35] P. Ryan and T. Peacock. Opacity – Further Insights on an Information Flow Property. *Technical Report Series – University of Newcastle Upon Tyne Computing Science*, 958, 2006.

[36] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[37] D. Sutherland. A model of information. In *Proceedings of the 9th National Computer Security Conference*, volume 247, 1986.

[38] J. Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems (TODS)*, 10(3):289–321, 1985.

[39] W. Winsborough and N. Li. Safety in automated trust negotiation. *ACM Transactions on Information and System Security (TISSEC)*, 9(3), 2006.

[40] W. H. Winsborough, K. E. Seamons, and V. E. Jones. Automated trust negotiation. In *DARPA Information Survivability Conference and Exposition*, volume 1, 2000.

[41] S. Zdancewic and A. Myers. Robust declassification. In *IEEE Computer Security Foundations Workshop*, pages 15–23, 2001.