Information leakage in Datalog-based trust management systems

Moritz Y. Becker
Microsoft Research
Cambridge, UK

Masoud Koleini
School of Computer Science
University of Birmingham, UK

February 2011

Technical Report
MSR–TR–2011–11

Microsoft Research
Roger Needham Building
7 J.J. Thomson Avenue
Cambridge, CB3 0FB
United Kingdom

Information leakage in Datalog-based trust management systems

Moritz Y. Becker
Microsoft Research
Cambridge, UK

Masoud Koleini
School of Computer Science
University of Birmingham, UK

February 2011

Abstract

Most trust management systems that specify authorization in a high-level policy language and support credential-based authorization are vulnerable to a little known class of attacks, so-called probing attacks. A probing attack is conducted by running probes against the system, i.e., by submitting access requests together with credentials, and observing the system's reactions. This may enable an external adversary to gain knowledge about confidential facts in the policy. We present the first complete decision procedure for checking if an adversary, characterized by a set of probes available in an attack, is unable to gain knowledge about confidential information about a policy specified in Datalog (in which case the information is said to be opaque). This also positively answers the hitherto open question of whether the opacity problem in this setting is decidable. We describe a prototype implementation that is equipped with a number of optimizations to prune the search space, and empirical results from experiments, based on a realistic delegation policy, to test the scalability of the algorithm and the effectiveness of our optimization methods.

1 Introduction

In decentralized trust management systems [6], the authorization rules are specified in a machine-enforceable high-level *policy language* such as XACML [26], Ponder [12], SecPAL [4] or RT [23] – just to name a few. Users requesting access may submit *credentials* to support their request, and access is granted only if the request complies with the policy in conjunction with the submitted credentials.

There is a class of attacks on such systems, called *probing attacks*, that has received attention only very recently [3]. In a probing attack, the adversary submits a series of access requests together with supporting credentials, so-called *probes*, to a service, and, by observing the service's reactions, gains knowledge about confidential information in the service's policy.

Here is a simple example of a probing attack on a SecPAL policy. (The attack can be straightforwardly adapted to other languages.)

The service `Hospital` has an authorization policy containing the publicly readable assertions

> `Hospital` **says** x canCreatePatientAccount **if**
> x isOver18.
> `Hospital` **says** `AgeCert` **cansay** x isOver18.

The first assertion states that any principal x can create a patient account if (`Hospital` says that) x is an adult. The second assertion *delegates authority* over the isOver18 predicate to `AgeCert`. In other words, if `AgeCert` says that someone is an adult, then `Hospital` will also say so.

`Hospital`'s policy also contains confidential assertions that are not visible to the adversary `Eve`, for instance, whether `Bob` is a patient or not. `Eve` submits two probes:

1. `Eve` collaborates with `AgeCert`, and hence can get hold of two `AgeCert`-issued credentials "`AgeCert` **says** `Eve` isOver18 **if** `Bob` isPatient" and "`AgeCert` **says** `Hospital` **cansay** `Bob` isPatient".

   In the first probe, she submits both credentials together with the access request "`Hospital` **says** `Eve` canCreatePatientAccount?". The service evaluates the query against its policy *in union* with the two submitted credentials, and responds by granting access.

2. In the second probe, `Eve` submits only the second of the two credentials, together with the same access request. This time the service denies access.

From these two probes, `Eve` observes that the first of the two credentials is crucial in evaluating the query. But this is only possible if `Hospital` **says** `Bob` isPatient. She has therefore *detected* a confidential fact through probing: the fact holds in all policies that behave in the same observable way with respect to the probes.

It is common for authorization policies to contain confidential information, and it takes little effort to conduct a probing attack; at the same time, it is non-trivial to see, in more involved examples, if a piece of information can be detected through probing or not. It is therefore critical to have an automated method for verifying non-detectability – or, *opacity* – of confidential information in credential systems.

Becker [3] presents a deduction system for proving detectability in Datalog-based policies. However, this system cannot be easily turned into an automated method, and is not provably complete, and so it cannot be used to prove the more useful safety property of opacity. To tackle opacity, Becker proposes a modification to existing Datalog-based policy languages, which guarantees opacity of all policy facts apart from those explicitly specified as detectable. The problem with this approach is that modifying the deployed policy language in a distributed system is often not possible or feasible.

In the current paper, we take a different, more analytical, approach to dealing with opacity. We first present a formal framework for probing attacks in credential systems (Section 3). It is based on the notions of detectability and opacity in terms of observational equivalence, similar as in [3], but is more general and applies to a wider range of policy

languages including XACML [26] and DKAL2 [19]. We illustrate these concepts using the example of a realistic delegation-based data sharing policy specified in Datalog (Section 4).

The primary technical contribution of this paper is an algorithm for checking if a given query is opaque in a given policy to an adversary specified by a set of probes (Section 5). The algorithm works on policies written in Datalog, which is the basis of many existing policy languages. The algorithm is not only sound, but also complete and terminating: if it fails to prove opacity (in which case failure is reported in finite time), then the given fact is provably detectable. This is a strong result, as the existence of a complete decision procedure for opacity in this context is far from obvious.

Another attractive feature of the algorithm is its constructiveness. Intuitively, a property is opaque in a policy if there exists some policy that behaves in the same way as the first policy with regards to the adversary's probes, but in which the property does not hold. If the property is opaque, the algorithm actually constructs such a "witness" policy. In fact, it can iteratively constuct a finite sequence of such witnesses that subsumes the generally infinite set of *all* witnesses. What does this feature buy us? Without it, the algorithm would be merely *possibilistic*: the mere existence of a witness policy, no matter how pathological it may be, would be sufficient for opacity. But since the algorithm constructs the witnesses, they can be assigned probabilities (or the security analyst could interactively discard unlikely witnesses). The final result could therefore be interpreted as the *probability* (or an informal degree of likelihood) of the property being opaque.

We describe in Section 6 the implementation of a prototype based on the algorithm, which employs a number of methods for cutting the high computational cost by pruning the search space. We present in Section 7 experimental results from applying the prototype to a variety of test cases based on the delegation policy from Section 4. The results show that medium-sized opacity verification problems become feasible with the optimizations turned on, whereas the naive implementation of the algorithm is usable only for very small test cases.

Limitations of our approach and future work are discussed in Section 8.

The proofs for our theoretical results are quite involved. We omit them here due to lack of space, but include the essential lemmas. Full proofs will appear in a technical report.

2 Related work

Probing attacks on credential systems were first mentioned by Gurevich and Neeman [18]. One of the primary design goals of their policy language, DKAL, was to provide protection against probing attacks. However, they do not precisely define what they are protecting against, and indeed, it has been shown that DKAL2 [19] is susceptible to probing attacks [3].

The first formal framework for reasoning about probing attacks was published by Becker [3]. It provides definitions of detectability and opacity, based on observational equivalence between policies. However, it makes the assumption that policies are unstructured sets of assertions, and that the submitted credentials are also assertions that are simply added to the local policy during evaluation. This assumption is valid for many logic-based policy languages such as SecPAL [4], RT [23], Cassandra [5], SD3 [22], and

Binder [13], but not for languages such as XACML [26] or Ponder [12], in which policies have some (e.g. hierarchical) structure, or languages such as DKAL, where incoming credentials are filtered and transformed before being added to the policy. In contrast, our abstract framework presented in Section 5 makes no such assumptions, giving rise to more generally applicable definitions of detectability and opacity.

The paper by Becker describes an inference system for proving detectability in Datalog-based policies, which can be used for proving that the adversary can get hold of confidential information. However, it cannot be easily implemented as an automated reasoning procedure, as it contains some nonconstructive elements. Moreover, it is not complete, and hence cannot be used to prove the more useful property of opacity, namely that the adversary *cannot* detect a piece of information. The algorithm described in Section 5 is the first decision procedure for proving opacity (or detectability).

The information flow property of opacity has been studied in the contexts of cryptographic protocols [24] and general state transition systems [8], where it is closely related to non-inference [27] and non-deducibility [29]. Our definition of detectability, the negation of opacity, bears some resemblance to the knowledge operator K_i in epistemic modal logic [15], which has been used for reasoning about secrecy in stateful, temporally evolving multi-agent systems [20].

Research on information flow has mainly focused on stateful, temporal computations [28]. The current setting is quite different as there is no notion of state, state transition, run or trace, and, most importantly, probes may contain credentials that are temporarily combined with the local policy during query evaluation – this is precisely what makes the analysis so hard. In contrast, the adversaries considered in computational information flow analysis typically cannot inject code into the program.

A policy could be seen as a database, and the probes as queries against the database. Detectability through probing could therefore be seen as related to the database inference problem, which is concerned with covert channels through which confidential information from a database can leak to a database user. A wide variety of such channels have been studied [21, 16], mainly for relational databases. Bonatti et al. have studied the database inference problem in deductive databases [7], which are similar to the Datalog-based policies considered in the current paper. However, the problem considered in the current paper is harder, as it corresponds to users who can temporarily inject new rules and relations into the database (which is not natural in the typical database context).

The problem of constructively proving opacity in Datalog policies is somewhat related to Inductive Logic Programming (ILP) [25]. The opacity problem can be reduced to the problem of constructing a policy, which is a set of clauses, that grants access for a given set of "positive" probes, and that denies access for a second given set of "negative" probes. Similarly, the ILP problem could be described as that of constructing a set of clauses that implies a given set of "positive" experiments, and that is inconsistent with a second given set of "negative" experiments. But again, the main difference is that the probes we consider are much harder than the experiments considered in ILP, which are simple clauses or even just atoms. In contrast, a probe has the expressive power of a goal in Hypothetical Logic Programming (HLP) [14]: in HLP, a goal may be supplemented by a set of "hypothetical" clauses which are temporarily added to the logic program (corresponding to our submitted credentials). Hence opacity checking could be framed as an induction problem on a hypothetical logic program.

3 A framework for probing attacks

3.1 Abstract framework

This section establishes the fundamental concepts for reasoning about probing attacks in credential systems. We will give an abstract definition of policy language and probes, and based on that, define the complementary notions of opacity and detectability.

Definition 3.1 (Policy language, probe)**.** A *policy language* is a triple $(\mathbf{Pol}, \mathbf{Prb}, \vdash)$, where **Pol** and **Prb** are sets called *policies* and *probes*, respectively, and $\vdash$ is a binary infix relation from $\mathbf{Pol} \times \mathbf{Prb}$, called *decision relation*.

Let $A \in \mathbf{Pol}$, $\pi \in \mathbf{Prb}$. If $A \vdash \pi$ we say that π is *positive in* A; otherwise (i.e., $A \nvdash \pi$), π is *negative in* A.

This definition is more abstract and general than the one given by Becker [3], as it does not make any assumptions about the structure of policies and probes. In particular, it encompasses policy languages where a policy is not simply a flat collection of assertions, or where the submitted credentials within a probe are not simply added to the local policy for evaluation.

Although Definition 3.1 does not prescribe the structure of probes, it helps to think of a probe as a pair containing a set of credentials that the adversary submits to the service under attack, and a query corresponding to some access request. A positive probe is one that leads to an access grant, whereas a negative leads to an access denial.

To illustrate Definition 3.1, we briefly sketch how it would be instantiated to the concrete policy languages SecPAL [4], DKAL2 [19], and XACML [26].

In SecPAL, a policy is a set of SecPAL *assertions* such as "`Alice` **says** x canRead **if** x canWrite" or "`Bob` **says** `Eve` **cansay** x canWrite". Access requests are mapped to SecPAL *queries*, which are first-order formulas over atoms of the form "⟨*Principal*⟩ **says** ⟨*Fact*⟩". An inference system defines which queries are deducible from a policy. A user's access request is mapped to a query, and is granted only if the query is deducible from the *union* of the local policy *and* the set of credentials (which are also just assertions) submitted by the user together with the request. Therefore, a probe π is naturally defined as a pair $\langle A, \varphi \rangle$ containing a set A of credentials and a query φ. Then $A_0 \vdash \pi$ iff φ is deducible from $A_0 \cup A$. The definitions can be instantiated in a very similar fashion for other related languages such as RT [23], Cassandra [5], SD3 [22], and Binder [13].

In DKAL2, a policy is a set of so-called *infon terms* such as "`Eve` **said** `Alice` canRead" or "`Bob` **implied** `Alice` **said** `Eve` canWrite". As in SecPAL, a set of inference rules defines which other infon terms can be deduced from a policy. However, infon terms sent by the adversary (corresponding to submitted credentials) are not simply added verbatim to the local policy, but are converted depending on the term's shape. For example, the infon term "`A` canRead ⟵ `B` canWrite" submitted by `Eve` would be imported as the infon term "`B` canWrite → `Eve` **implied** `A` canRead". Why this is done and what this means is beyond the scope of this paper; the important point here is that there are credential systems where the access query is not simply evaluated against the union of the local policy and the submitted credentials, but where the latter are first modified according to some rules.

In XACML, a policy (as in Definition 3.1) would correspond to a *PolicySet*, which is a hierarchical structure containing other PolicySets or items called (XACML) *Policy*. The

latter is a collection of *Rules*. XACML is thus an example of a system where a policy is not just a flat collection of assertions. Just as in DKAL2, incoming credentials (SOAP messages conveying *Attributes* [1]) may need to be transformed before being added to the policy. For example, the client-supplied Attribute may be written in SAML and would first need to be converted by an XACML Context Handler [2]. As in the case of DKAL2, the instantiation of $\vdash$ would have to take such transformations into account.

We now consider the adversary, i.e., the principal who mounts the probing attack against a service's policy A_0. Informally, the adversary has a *passive* and an *active* capability. A passively acting adversary only reads the visible part of A_0 that is presented to her by the service. An active adversary can additionally evaluate probes against A_0 and observe whether they are positive or negative in A_0. Typically, the adversary does not have the power to evaluate arbitrary probes, but only the probes *available* to her.

In the standard case where probes are pairs containing a set of credentials and a query, the availability of a probe is determined, firstly, by which credentials the adversary possesses or can freshly create, and, secondly, by which queries the service allows her to run. For instance, SecPAL services define an Authorization Query Table (AQT) that map access requests to SecPAL queries, so only these queries can be evaluated by clients. In DKAL2, incoming infon terms (corresponding to credentials) are filtered by a filtering policy, so not all credentials possessed by clients are available in probes. Our definition of available probes abstracts away such language-dependent details.

Definition 3.2 (Alikeness and available probes)**.** An *adversary* is defined by an equivalence relation $\simeq \subseteq \mathbf{Pol} \times \mathbf{Pol}$, and a set $\mathbf{Avail} \subseteq \mathbf{Prb}$ of *available probes*. If $A_1 \simeq A_2$ for two policies A_1 and A_2, we say that A_1 and A_2 are *alike*.

The alikeness relation, which specifies the adversary's passive capability, is also kept abstract in Definition 3.2. Typically, a policy can be split into a publicly visible and a private part (relative to a particular adversary). A useful instantiation in this case would be that two policies are alike iff their visible parts are syntactically equal. Alternatively, one could adopt a more semantic instantiation, such that two policies are alike iff their visible parts are semantically equivalent.

We can now define the adversary's *active* capability, again as an equivalence relation.

Definition 3.3 (Observational equivalence)**.** Two policies A_1 and A_2 are *observationally equivalent* ($A \equiv A'$) iff

1. $A_1 \simeq A_2$, and
2. $\forall \pi \in \mathbf{Avail}, A_1 \vdash \pi \iff A_2 \vdash \pi$.

Alikeness and observational equivalence induce two different notions of indistinguishability of policies. A passive adversary cannot distinguish policies that are alike. An active adversary can see the visible parts of a policy *and* run probes against it. These two capabilities are represented by conditions 1) and 2), respectively, in Definition 3.3. Hence an active adversary cannot distinguish policies that are observationally equivalent.

This paper focuses on the information flow property of opacity (and its negation, detectability). For the sake of completeness, we first show the definition of the coarser and less useful, but better-known, property of *non-interference* [17, 31]. In the context

of state transition systems, non-interference can be informally described as the system property that changing the high inputs does not affect the low outputs. Similarly, we say that a policy has the non-interference property if changing its non-visible parts does not affect the outcomes of the probes available to the adversary.

Definition 3.4 (Non-interference)**.** A policy $A \in \mathbf{Pol}$ has the *non-interference property* iff

$$\forall A' \in \mathbf{Pol} : \; A \simeq A' \Rightarrow A \equiv A'.$$

The main problem with non-interference is that it is too strict. It rules out many reasonable policies, in which the outcomes of probes may depend on the non-visible parts of the policy, but in such a way that the adversary cannot detect the nature of the dependency. What we are actually interested in is whether the adversary can infer, with certainty, that some property Φ holds about policy A, just by looking at the policy's public parts and by running the probes available to her. If she can, then we say that Φ is detectable in A, otherwise Φ is opaque in A. This is formalized in the following definition, which again is implicitly relative to a given adversary.

Definition 3.5 (Detectability, opacity)**.** A predicate $\Phi \subseteq \mathbf{Pol}$ is *detectable in* $A \in \mathbf{Pol}$ iff

$$\forall A' \in \mathbf{Pol} : \; A \equiv A' \Rightarrow \Phi(A').$$

A predicate $\Phi \in \mathbf{Pol}$ is *opaque in* $A \in \mathbf{Pol}$ iff it is not detectable in A, or equivalently, iff

$$\exists A' \in \mathbf{Pol} : \; A \equiv A' \; \wedge \; \neg\Phi(A').$$

By passively and actively observing a policy A, the adversary gains partial knowledge about the policy. More precisely, she can narrow down the set of policies she considers as possible candidates to the set of all A' such that $A \equiv A'$. If Φ holds in all these "possible" policies, then she can effectively detect Φ in A.

Our definition of detectability is closely related to the knowledge operator K_i in epistemic modal logic [15]: 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.

Opacity is the negation of detectability. The adversary cannot be sure that Φ holds in A if there exists at least one policy that she considers as possible, i.e., one that is observationally equivalent to A, in which Φ does not hold. Opacity thus requires the existence of a policy that mimics A in all observable respects, but in which Φ is false.

This definition is closely related to the (informal) definition of opacity in state transition systems [24, 8], where a predicate over a system trace is opaque if for all traces there exists a trace that is indistinguishable from the former one (from the adversary's point of view), in which the predicate is false.

From Definition 3.5 it follows that if Φ is detectable in A, then $\Phi(A)$ holds. Conversely, if $\Phi(A)$ does not hold, then Φ is (trivially) opaque in A.

The definitions established in this section so far provide a general framework for reasoning about probing attacks in credential systems. For particular trust management

frameworks, the definitions of policy language and alikeness need to be instantiated accordingly. We do this in the remainder of this section for Datalog. In Section 4, we discuss an example policy in Datalog, which will also help illustrate the definitions above.

3.2 Datalog-based policies

Datalog [30, 10, 9] is not used as a policy language per se, but is the semantic basis for many existing policy languages, and many others can be translated into Datalog, including SecPAL [4], RT [23], Cassandra [5], SD3 [22], and Binder [13]. Reasoning techniques and analysis tools for Datalog therefore apply to a wide range of policy languages. Datalog can be seen as pure Prolog (without cut, negation, updates etc.) without function symbols or, equivalently, as function-free first-order Horn clause logic.

The language is parameterized by a function-less first-order signature containing a countable set of predicate names and a countable set of constants. This gives rise to *atoms* P of the form $p(\vec{e})$, where p is a predicate symbol and $\vec{e}$ a sequence of *expressions* (i.e., first-order variables or constants) of p's arity.

The central construct in Datalog is a *clause*. A clause a is of the form

$$P_0 \leftarrow P_1, ..., P_n,$$

where $n \geq 0$, and the atom P_0 is called the *head* and the sequence of atoms $\vec{P} = \langle P_1, ..., P_n \rangle$ the *body*. The arrow $\leftarrow$ is usually omitted if $n = 0$. We write $\mathbf{Cls}$ to denote the set of all clauses. We write $\mathbf{hd}(a)$ to denote a's head and $\mathbf{bd}(a)$ to denote its body. Given a set of clauses A, we write $\mathbf{hds}(A)$ to denote the atom set $\{\mathbf{hd}(a) \mid a \in A\}$.

A *query* φ is either $\mathbf{true}$, $\mathbf{false}$ or a ground (i.e., variable-free) boolean formula (i.e., involving connectives $\neg$, $\wedge$ and $\vee$) over atoms. We write $\mathbf{Qry}$ to denote the set of all queries.

A query φ can evaluate to true or false (i.e., deducible or not deducible) in a set A of assertions. If φ evaluates to true, we write $A \vdash \varphi$, and otherwise $A \nvdash \varphi$ or, equivalently, $A \vdash \neg\varphi$. (We use the same symbol $\vdash$ for the Datalog evaluation relation as for the decision relation in Definition 3.1. As we shall see, the two are closely related.)

The Datalog evaluation relation $\vdash$ can be defined equivalently in various ways. Perhaps the most intuitive definition is by ways of a set of inference rules: $\vdash$ is the smallest relation such that the following holds.

- $A \vdash \mathbf{true}$.
- $A \vdash P_0$, if there exist atoms $P_1, ..., P_n$ (for some $n \geq 0$) such that $P_0 \leftarrow P_1, ..., P_n$ is a ground instance of some clause in A and $A \vdash P_i$ for all $i \in \{1, ..., n\}$.
- $A \vdash \neg\varphi$, if $A \vdash \varphi$ does not hold.
- $A \vdash \varphi \wedge \varphi'$, if $A \vdash \varphi$ and $A \vdash \varphi'$.
- $A \vdash \varphi \vee \varphi'$, if $A \vdash \varphi$ or $A \vdash \varphi'$.

Definition 3.6 (Consequence operator)**.** 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, let γ be a ground substitution (a total mapping from variables to constants).

$$\mathbf{T}_A(S) = \{\ P'_0\ \mid \exists\gamma\ \exists(P_0 \leftarrow P_1, ..., P_n) \in A, \\ \gamma(\{P_1, ..., P_n\}) \subseteq S, \\ P'_0 = \gamma(P_0)\ \}$$

The following definition introduces a number of terms that are fundamental to the algorithm described in Section 5. Lemma 3.8 establishes an important connection between probes and clause containment.

Definition 3.7 (Monotonicity, containment, equivalence)**.** A query is *monotonic* iff it is equivalent to one without negation. A probe $\langle A, \varphi\rangle \in \mathbf{Prb}$ is *monotonic* iff φ is monotonic.

A policy A is *contained in* a policy A' (we write: $A \preceq A'$) iff for all ground atoms P and all sets S of ground atoms: $A \vdash \langle S, P\rangle \Rightarrow A' \vdash \langle S, P\rangle$.

Two policies A and A' are *equivalent* (we write: $A \doteq A'$) iff $A \preceq A' \wedge A' \preceq A$.

Lemma 3.8. Let $A \subseteq \mathbf{Cls}$, $\vec{P}$ be a set of ground atoms and P a ground atom. $A \vdash \langle \vec{P}, P\rangle$ iff $\{P \leftarrow \vec{P}\} \preceq A$.

Proof.

($\Leftarrow$) This direction is straightforward.

($\Rightarrow$) Suppose the contrary. Let $n > 0$ be the smallest integer such that there exists a set S of ground atoms and a ground atom Q with $Q \in \mathbf{T}^n_{\{P\leftarrow\vec{P}\}\cup S}(\emptyset)$ and $Q \notin \mathbf{T}^\omega_{A\cup S}(\emptyset)$. Then there must be a ground instance $Q \leftarrow \vec{Q}$ of a clause in $\{P \leftarrow \vec{P}\} \cup S$ such that $\vec{Q} \subseteq \mathbf{T}^{n-1}_{\{P\leftarrow\vec{P}\}\cup S}(\emptyset) \subseteq \mathbf{T}^\omega_{A\cup S}(\emptyset)$. This clause cannot be in S, or else $Q \in \mathbf{T}^n_{A\cup S}(\emptyset)$.

Therefore, the ground instance is $P \leftarrow \vec{P}$, and thus $\vec{P} \subseteq \mathbf{T}^{n-1}_{\{P\leftarrow\vec{P}\}\cup S}(\emptyset) \subseteq \mathbf{T}^\omega_{A\cup S}(\emptyset)$. Hence $\mathbf{T}^\omega_{A\cup S}(\emptyset) = \mathbf{T}^\omega_{A\cup S\cup\vec{P}}(\emptyset)$. But from $A \vdash \langle \vec{P}, P\rangle$ it follows that $P \in \mathbf{T}^\omega_{A\cup S\cup\vec{P}}(\emptyset)$, and hence $P = Q \in \mathbf{T}^\omega_{A\cup S}(\emptyset)$, which contradicts the initial assumption.

□

Now we can instantiate the abstract Definitions 3.1 and 3.2. For evaluating probes, we adopt the simple model where the query of a probe is evaluated against the union of the service's policy and the credentials (i.e., clauses) of the probe.

Definition 3.9 (Datalog instantiation)**.** We instantiate **Pol** to the powerset of clauses, $\wp(\mathbf{Cls})$. A (Datalog) *policy* is hence a set $A_0 \subseteq \mathbf{Cls}$.

A (Datalog) *probe* π is a pair $\langle A, \varphi\rangle$, where $A \subseteq \mathbf{Cls}$ and $\varphi \in \mathbf{Qry}$. Hence **Prb** is instantiated to the set of all such probes. A probe is *ground* iff it does not contain any variables. We write $\neg\langle A, \varphi\rangle$ to denote the probe $\langle A, \neg\varphi\rangle$.

The *decision relation* $\vdash \subseteq \mathbf{Pol} \times \mathbf{Prb}$ is defined as follows:

$$A_0 \vdash \langle A, \varphi\rangle \iff A_0 \cup A \vdash \varphi.$$

A simple corollary of this definition is that $A \vdash \varphi$ is equivalent to $A \vdash \pi$, where $\pi = \langle\emptyset, \varphi\rangle$ is a probe containing no submitted credentials.

Definition 3.10 (Adversary, Datalog alikeness)**.** An *adversary* is defined by a set $\mathbf{Avail} \subseteq \mathbf{Prb}$ and a unary predicate $\mathbf{Visible} \subseteq \mathbf{Cls}$. If $\mathbf{Visible}(a)$ for some $a \in \mathbf{Cls}$, we say that a is *visible*.

We extend **Visible** to policies by defining the *visible part* of A, $\mathbf{Visible}(A)$, as $\{a \in A \mid \mathbf{Visible}(a)\}$, for all $A \subseteq \mathbf{Cls}$.

Two policies $A_1, A_2 \subseteq \mathbf{Cls}$ are *alike* ($A_1 \simeq A_2$) iff $\mathbf{Visible}(A_1) = \mathbf{Visible}(A_2)$.

The definition of alikeness for Datalog is a very syntactic one. An alternative, more semantic, condition would be to require $\mathbf{Visible}(A_1) \vdash \pi \iff \mathbf{Visible}(A_2) \vdash \pi$ for all $\pi \in \mathbf{Prb}$. We decided against this definition because it would render alikeness undecidable (this follows from the undecidability of the containment problem in Datalog [10]). Moreover, it seems a reasonable assumption that adversaries can passively distinguish between two syntactically different visible clause sets even if they are semantically equivalent, e.g., $\{p \leftarrow q.,\ q.\}$ and $\{p.,\ q.\}$. (In sets, we mark the end of a clause by a period '.' to avoid ambiguity.)

Definitions 3.9 and 3.10 induce instantiations for the Datalog definitions of observational equivalence between policies, and of opacity and detectability. Recall that the latter two were defined for arbitrary properties of policies. Here, we are interested in a particular class of policy properties, namely whether a given probe is positive.

Definition 3.11 (Probe detectability & opacity)**.** A probe $\pi \in \mathbf{Prb}$ is *detectable in* $A \in \mathbf{Pol}$ iff

$$\forall A' \in \mathbf{Pol} :\ A \equiv A' \Rightarrow A' \vdash \pi.$$

A probe $\pi \in \mathbf{Prb}$ is *opaque in* $A \in \mathbf{Pol}$ iff it is not detectable in A, or equivalently, iff

$$\exists A' \in \mathbf{Pol} :\ A \equiv A' \ \wedge\ A' \nvdash \pi.$$

Note that this definition is just a specialization of Definition 3.5, with the predicate Φ instantiated to $\{A \subseteq \mathbf{Cls} \mid A \vdash \pi\}$.

4 Example: Delegation policy

We illustrate the definitions from the previous section using a realistic example of an authorization policy written in Datalog. The example also serves as the basis for the test cases in Section 7.

Our example is taken from a grid computing scenario. A compute cluster allows users to run compute jobs. The execution of a job may require read access to data that is stored in an external data center. The cluster has a policy that governs who can run compute jobs, and the data center has a policy that governs who can access data. Both policies *delegate authority* over certain attributes to trusted third parties.

In the following, we adopt the convention that the first parameter of a predicate denotes the principal "saying" (i.e., vouching for) the predicate, and the second parameter denotes the subject of the predicate. Variables are written in lower case and *italics*, and constants are capitalized and written in `typewriter` font. For instance, $\mathsf{canExec}(\texttt{Cluster}, x, j)$ intuitively means that `Cluster` says that x can execute job j.

The first clause stipulates that anyone who is a member and owns a job (according to `Cluster`) can execute that job (according to `Cluster`), if the data center `Data` allows `Cluster` to read the data associated with that job.

$$\begin{aligned}&\mathsf{canExec}(\texttt{Cluster}, x, j) \leftarrow \\ &\quad \mathsf{isMem}(\texttt{Cluster}, x), \\ &\quad \mathsf{owns}(\texttt{Cluster}, x, j), \\ &\quad \mathsf{canRead}(\texttt{Data}, \texttt{Cluster}, j).\end{aligned} \tag{1}$$

`Cluster` delegates authority over job ownership to trusted third parties. Hence if a third party trusted by `Cluster` says that x owns job j, then `Cluster` is also willing to say that x owns j.

$$\begin{aligned}&\mathsf{owns}(\texttt{Cluster}, x, j) \leftarrow \\ &\quad \mathsf{owns}(y, x, j), \mathsf{isTTP}(\texttt{Cluster}, y).\end{aligned} \tag{2}$$

`Cluster` also delegates authority over membership to trusted third parties.

$$\begin{aligned}&\mathsf{isMem}(\texttt{Cluster}, x, j) \leftarrow \\ &\quad \mathsf{isMem}(y, x, j), \mathsf{isTTP}(\texttt{Cluster}, y).\end{aligned} \tag{3}$$

The next clause implements a variant of discretionary access control. The data center `Data` stipulates that owners y of data associated with job j can delegate read access to this data to other principals x.

$$\begin{aligned}&\mathsf{canRead}(\texttt{Data}, x, j) \leftarrow \\ &\quad \mathsf{canRead}(y, x, j), \mathsf{owns}(\texttt{Data}, y, j).\end{aligned} \tag{4}$$

Just like `Cluster`, `Data` also delegates authority over ownership to third parties it trusts.

$$\begin{aligned}&\mathsf{owns}(\texttt{Data}, x, j) \leftarrow \\ &\quad \mathsf{owns}(y, x, j), \mathsf{isTTP}(\texttt{Data}, y).\end{aligned} \tag{5}$$

Finally, both `Cluster` and `Data` specify certificate authority `CA` as a trusted third party.

$$\mathsf{isTTP}(\texttt{Cluster}, \texttt{CA}). \tag{6}$$

$$\mathsf{isTTP}(\texttt{Data}, \texttt{CA}). \tag{7}$$

`Cluster` has an interface that allows users to submit a job execution request. When some user `U` requests to execute a job `J`, the corresponding query

$$\mathsf{canExec}(\texttt{Cluster}, \texttt{U}, \texttt{J}) \tag{8}$$

is evaluated against the policy consisting of the clauses shown above, in union with the (possibly empty) set of credentials submitted by the user together with the request.

Now consider a user, `Eve`, who plays the role of the adversary in our scenario. She possesses two credentials issued by `CA`, one that asserts her ownership over job `Job`, and one that asserts her membership status.

$$\mathsf{owns}(\texttt{CA}, \texttt{Eve}, \texttt{Job}). \tag{9}$$

$$\mathsf{isMem}(\texttt{CA}, \texttt{Eve}). \tag{10}$$

Eve can also self-issue credentials. She issues one stating that Cluster is permitted to read data associated with Job:

$$\mathsf{canRead}(\texttt{Eve}, \texttt{Cluster}, \texttt{Job}). \tag{11}$$

Eve is interested in finding out if Bob is a member, according to Cluster's policy. Of course, she does not have the authority to query this fact directly, so instead she self-issues a credential with which she hopes to be able to detect this fact through probing:

$$\mathsf{canRead}(\texttt{Eve}, \texttt{Cluster}, \texttt{Job}) \leftarrow \tag{12}$$
$$\quad \mathsf{isMem}(\texttt{Cluster}, \texttt{Bob}).$$

This credential states that she is willing to give Cluster read access, provided that Bob is a member of Cluster.

Let A_0 be the policy consisting of clauses (1)–(7), and $A_{\texttt{Eve}}$ be the set of clauses (9)–(12). Let $\varphi_{\texttt{Eve}}$ be the query (8) instantiated by Eve's request to execute Job on Cluster:

$$\varphi_{\texttt{Eve}} = \mathsf{canExec}(\texttt{Cluster}, \texttt{Eve}, \texttt{Job}).$$

$A_{\texttt{Eve}}$ and $\varphi_{\texttt{Eve}}$ together give rise to a set of $2^4 = 16$ available probes that Eve is able to run against A_0:

$$\mathbf{Avail} = \{\langle A, \varphi_{\texttt{Eve}}\rangle \mid A \subseteq A_{\texttt{Eve}}\}.$$

For simplicity, we assume that $\mathbf{Visible} = \emptyset$, i.e., Eve is not able to passively read any of the clauses in A_0.

Based on this scenario, we make the following observations.

1. $A_0 \vdash \langle A_{\texttt{Eve}}, \varphi_{\texttt{Eve}}\rangle$, in other words, $A_0 \cup A_{\texttt{Eve}} \vdash \varphi_{\texttt{Eve}}$. The derivation goes roughly as follows: Credential (9) proves Eve's ownership over Job to both Data and Cluster. Hence Data allows Eve to delegate read access to Cluster using (11). Furthermore, (10) is sufficient for proving Eve's membership to Cluster, hence all body atoms of (1) are satisfied, which implies $\varphi_{\texttt{Eve}}$.

2. $A_0 \vdash \langle\{(9)\text{–}(11)\}, \varphi_{\texttt{Eve}}\rangle$, which follows from the fact that the derivation in item 1) above only makes use of the clauses (9)–(11). In fact, this is a *minimally positive* probe in the sense that all probes with strictly smaller credential sets are negative.

3. $A_0 \nvdash \langle A, \varphi_{\texttt{Eve}}\rangle$, for all $A \subseteq \{(9),(10),(12)\}$. In particular, replacing clause (11) in the probe in item 2) above by clause (12) produces a negative probe. Note that the two clauses only differ in the body.

4. All policies A_0' that are observationally equivalent to A_0, i.e., that exhibit the same behaviour as stated in items 1)–3) above, satisfy the property that $A_0 \nvdash \mathsf{isMem}(\texttt{Cluster}, \texttt{Bob})$. For suppose the contrary were the case. From item 2), we know that $\varphi_{\texttt{Eve}}$ holds in $A_0 \cup \{(9)\text{–}(11)\}$. By assumption, the body of clause (12) is true in A_0, which means that replacing clause (11) by (12) in the probe cannot make a difference. But this contradicts item 3), which states that $\varphi_{\texttt{Eve}}$ does *not* hold in $\{(9),(10),(12)\}$.

5. It follows that the probe $\langle \emptyset, \neg \mathsf{isMem}(\texttt{Cluster}, \texttt{Bob}) \rangle$, which is not in **Avail**, is detectable in A_0. In other words, `Eve` can be sure that `Bob` is not a member of `Cluster`.

6. Here is an example of an opaque probe:

$$\pi = \langle \{(9),(11)\}, \mathsf{canRead}(\texttt{Data}, \texttt{Cluster}, \texttt{Job}) \rangle.$$

The opacity of π is non-trivial, as $A_0 \vdash \pi$ actually holds. But there exists a policy that is observationally equivalent to A_0 such that $A_0 \nvdash \pi$: just remove clause (4) from A_0 and replace `Data` by x in (1).

5 Verifying opacity

This section presents an algorithm for verifying opacity. Given a set of available probes, the algorithm decides if a given probe is opaque (or detectable) in a given Datalog policy.

The algorithm works with arbitrary input policies, but we restrict the input probes to ground ones, in order to simplify the problem. This restriction is reasonable, as attribute and delegation credentials are usually issued for one specific principal and purpose, and are thus ground anyway. Section 8 discusses this restriction in more detail.

How can opacity and detectability be checked? One apparent solution that may spring to mind would be to view the policy clauses as first order logic formulas, and to interpret $\vdash$ as implication. More precisely, the arrow $\leftarrow$ in a clause would be interpreted as reverse implication, and the commas as conjunction. A probe $\langle A, \varphi \rangle$ that is positive in the policy A_0 would be interpreted as $(\bigwedge A) \Rightarrow \varphi$. A negative probe $\pi = \langle A, \varphi \rangle$ can be converted into a positive one by replacing it by $\neg\pi = \langle A, \neg\varphi \rangle$. The conjunction of all available probes, interpreted thus, together with $\mathbf{Visible}(A_0)$, should then imply *all* queries that are detectable. Queries that are not implied would then be opaque.

However, this approach does not work in general, as a simple counterexample shows. Suppose $\langle \{p.\}, \neg q \rangle$ is positive in A_0, and is the only available probe. This probe would be interpreted, in the approach described above, as the formula $p \Rightarrow \neg q$, which does not imply $\neg q$, even though the latter is detectable in A_0.

Our algorithm for verifying opacity is more involved. Rather than presenting it at once, we will first describe the principles that lead to it one by one.

In the following, we assume as given a policy $A_0 \subseteq \mathbf{Cls}$, a ground probe $\pi_0 \in \mathbf{Prb}$, and an adversary defined by a set $\mathbf{Avail} \subseteq \mathbf{Prb}$ of ground probes and the visibility function **Visible**. (We will only be interested in $\mathbf{Visible}(A_0) \subseteq A_0$). The algorithm should decide if π_0 is opaque in A_0, relative to the adversary.

5.1 Opacity as existential problem

Recall that π_0 is opaque in A_0 iff there exists a policy A_0' that is observationally equivalent to A_0 (with respect to the probes in **Avail**), but such that π_0 is negative in A_0'. To prove opacity, we attempt to construct such an A_0'. Conversely, to prove detectability, we need to prove that no such A_0' exists.

5.2 Query decomposition

Consider a probe $\pi = \langle A, \varphi_1 \vee \varphi_2 \rangle \in \mathbf{Avail}$ that is positive in A_0. The search attempts to find a policy A_0' such that $A_0' \vdash \pi$ holds, in other words, $A_0' \cup A \vdash \varphi_1 \vee \varphi_2$. This is equivalent to either finding an A_0' such that $A_0' \vdash \langle A, \varphi_1 \rangle$, or finding one such that $A_0' \vdash \langle A, \varphi_2 \rangle$. A disjunction in the query of a probe in **Avail** therefore corresponds to a branch in the search for A_0'.

What about probes in **Avail** that are negative in A_0? Since $A_0 \nvdash \pi$ is equivalent to $A_0 \vdash \neg\pi$, we can convert all negative probes in **Avail** into equivalent positive ones before dealing with the disjunctions in positive probes.

The algorithm starts by computing all such disjunctive branches, the leaf nodes of which will form the initial states of the search. This is done by computing the disjunctive normal form of the converted queries in **Avail**, and then constructing a cartesian product over the disjuncts.

Definition 5.1 (Disjunctive normal form)**.** Let $\mathbf{dnf}(\varphi)$ denote the disjunctive normal form of the query φ, encoded as a set of pairs (S^+, S^-) of sets of atoms, so

$$\varphi \iff \bigvee_{(S^+,S^-)\in\mathbf{dnf}(\varphi)} \left(\bigwedge S^+ \wedge \neg \bigvee S^- \right).$$

Example 5.2. Let $\varphi = (p \wedge q \wedge \neg s) \vee (\neg p \wedge \neg q \wedge s)$. Then $\mathbf{dnf}(\varphi) = \{(\{p, q\}, \{s\}), (\{s\}, \{p, q\})\}$.

Lemma 5.3. Let $\langle A, \varphi \rangle \in \mathbf{Prb}$. Then $A_0 \vdash \langle A, \varphi \rangle$ iff

$$\exists (S^+, S^-) \in \mathbf{dnf}(\varphi) : A_0 \vdash \langle A, \textstyle\bigwedge S^+ \rangle \text{ and } A_0 \nvdash \langle A, \bigvee S^- \rangle.$$

The function $\mathbf{flatten}_{A_0}$, defined below, first performs the mentioned conversion of negative probes into equivalent positive ones. It then constructs a set of pairs of probe sets; each such pair corresponds to a disjunctive search branch. Lemma 5.5 states the fundamental property of $\mathbf{flatten}_{A_0}$.

Definition 5.4 (Flatten)**.** Let $\Pi \subseteq \mathbf{Prb}$. Then $\mathbf{flatten}_{A_0}(\Pi)$ is a set of pairs (Π^+, Π^-) of sets of probes defined inductively as follows:

$$\mathbf{flatten}_{A_0}(\emptyset) = \{(\emptyset, \emptyset)\}.$$

$$\begin{aligned}
&\mathbf{flatten}_{A_0}(\Pi \cup \{\langle A, \varphi \rangle\}) = \{(\Pi^+, \Pi^-) \mid \\
&\quad \exists\, (\Pi_0^+, \Pi_0^-) \in \mathbf{flatten}_{A_0}(\Pi),\ (S^+, S^-) \in \mathbf{dnf}(\tilde{\varphi}) : \\
&\qquad \Pi^+ = \Pi_0^+ \cup \{\langle A, \textstyle\bigwedge S^+ \rangle\} \text{ and} \\
&\qquad \Pi^- = \Pi_0^- \cup \{\langle A, \textstyle\bigvee S^- \rangle\}, \\
&\quad \text{where } \tilde{\varphi} = \varphi \text{ if } A_0 \vdash \langle A, \varphi \rangle, \text{and } \tilde{\varphi} = \neg\varphi \text{ otherwise.}
\end{aligned}$$

Lemma 5.5. Let $A_0' \subseteq \mathbf{Cls}$ and $\Pi \subseteq \mathbf{Prb}$.

$$\begin{aligned}
&\forall \pi \in \Pi : A_0' \vdash \pi \iff A_0 \vdash \pi \text{ iff} \\
&\quad \exists (\Pi_0^+, \Pi_0^-) \in \mathbf{flatten}_{A_0}(\Pi) : \\
&\qquad (\forall \pi \in \Pi_0^+ :\ A_0' \vdash \pi) \text{ and } (\forall \pi \in \Pi_0^- :\ A_0' \nvdash \pi).
\end{aligned}$$

Proof. ($\Rightarrow$) Assume $\forall \pi \in \Pi : A_0' \vdash \pi \iff A_0 \vdash \pi$. For the sake of contradiction, suppose the negation of the right hand side were true. By the definition of **flatten**, there thus exists $\pi = \langle A_\pi, \varphi \rangle \in \Pi$ and $(S^+, S^-) \in \mathbf{dnf}(\tilde{\varphi})$ (where $\tilde{\varphi} = \varphi$ if $A_0 \vdash \pi$, and $\tilde{\varphi} = \neg\varphi$ otherwise) such that $A_0' \nvdash \langle A_\pi, \bigwedge S^+ \rangle$ or $A_0' \vdash \langle A_\pi, \bigvee S^- \rangle$.

If $A_0 \vdash \pi$, then it follows by Lemma 5.3 that $A_0' \nvdash \pi$, which contradicts the initial assumption. If $A_0 \nvdash \pi$, then by Lemma 5.3, we have $A_0' \nvdash \langle A_\pi, \neg\varphi \rangle$ and hence $A_0' \vdash \pi$, again contradicting the initial assumption. Therefore, the right hand side of the equivalence holds.

($\Leftarrow$) Assume the right hand side of the equivalence. Let $\pi = \langle A_\pi, \varphi \rangle \in \Pi$.

Suppose $A_0 \vdash \pi$. Then, by the assumption and the definition of **flatten**, there exists $(S^+, S^-) \in \mathbf{dnf}(\varphi)$ such that $A_0' \vdash \langle A_\pi, \bigwedge S^+ \rangle$ and $A_0' \nvdash \langle A_\pi, \bigvee S^- \rangle$. From Lemma 5.3 it follows that $A_0' \vdash \pi$, as required.

Now suppose $A_0 \nvdash \pi$. Then, by a similar argument, we can show that $A_0' \vdash \langle A_\pi, \neg\varphi \rangle$ and hence $A_0' \nvdash \pi$, as required. □

Lemma 5.5 shows that the problem of finding a policy A_0' that is observationally equivalent to A_0 can be reduced to finding an A_0' and picking a pair $(\Pi_0^+, \Pi_0^-) \in \mathbf{flatten}_{A_0}(\mathbf{Avail})$ such that all probes in Π_0^+ are positive, and all probes in Π_0^- are negative in A_0'. Furthermore, observational equivalence also requires $A_0' \simeq A_0$, in other words, $\mathbf{Visible}(A_0') = \mathbf{Visible}(A_0)$.

Example 5.6. Suppose $\mathbf{Avail} = \{\pi_1, \pi_2\}$, where $\pi_1 = \langle A_1, \neg p \vee q \rangle$ and $\pi_2 = \langle A_2, \neg p \vee q \rangle$. Suppose further that $A_0 \vdash \pi_1$ and $A_0 \nvdash \pi_2$. Let $\pi_1^p = \langle A_1, p \rangle$, $\pi_1^q = \langle A_1, q \rangle$, $\pi_2^p = \langle A_2, p \rangle$, and $\pi_2^q = \langle A_2, q \rangle$. Then $\mathbf{flatten}_{A_0}(\mathbf{Avail})$ contains four pairs of probe sets: $(\{\pi_2^p\}, \{\pi_1^p\})$, $(\emptyset, \{\pi_1^p, \pi_2^q\})$, $(\{\pi_1^q, \pi_2^p\}, \emptyset)$, and $(\{\pi_1^q\}, \{\pi_2^q\})$.

Apart from observational equivalence, opacity also requires that $\pi_0 = \langle A, \varphi \rangle$ be negative in A_0'. By Lemma 5.3, this is equivalent to picking a pair $(S^+, S^-) \in \mathbf{dnf}(\neg\varphi)$ such that $A_0' \vdash \langle A, \bigwedge S^+ \rangle$ and $A_0' \nvdash \langle A, \bigvee S^- \rangle$.

5.3 High-level overview

Summarizing the last observation, we have reduced the problem of proving opacity of π_0 in A_0 to finding an $A_0' \simeq A_0$ and picking $(\Pi_0^+, \Pi_0^-) \in \mathbf{flatten}_{A_0}(\mathbf{Avail})$ and $(S^+, S^-) \in \mathbf{dnf}(\neg\varphi)$ such that

1. all probes in $\Pi^+ = \Pi_0^+ \cup \{\langle A, \bigwedge S^+ \rangle\}$ are positive,
2. and all probes in $\Pi^- = \Pi_0^- \cup \{\langle A, \bigvee S^- \rangle\}$ are negative.

If such an A_0' exists, we say that A_0' is a *witness* (for the opacity of π_0 in A_0).

The search space for witnesses is infinite in general (since **Cls** is usually infinite), but we can construct A_0' in a more goal-directed way. Here is a high-level overview of the search strategy. Taking the requirement $A_0' \simeq A_0$ into account, the construction of a witness candidate starts off with the policy $\mathbf{Visible}(A_0)$. This policy will generally not satisfy requirement 1). To satisfy 1), we go through each probe in Π^+ one by one, and for each such probe, we need to add one or more clauses to the witness candidate. The monotonicity of $\vdash$ guarantees that adding clauses does not invalidate 1) for the

probes already considered. However, adding clauses may violate requirement 2), so each time clauses are added, we check if 2) still holds. If not, we need to backtrack to try out a different way of satisfying 1). When all probes in Π^+ have been considered, the constructed candidate is guaranteed to be a witness.

5.4 Preserving alikeness

Adding clauses to the candidate in order to satisfy 1) may violate the alikeness requirement $A'_0 \simeq A_0$, because we may be adding a clause $a \notin A_0$ such that $\mathbf{Visible}(a)$. To ensure that only invisible clauses are added, we assume that there exists a nullary predicate symbol p_{Hi} that neither occurs in A_0 nor in π_0 nor in **Avail**, such that for all $a \in \mathbf{Cls}$, $\neg\mathbf{Visible}(a)$ whenever p_{Hi} occurs in a. This is a reasonable assumption, as the service (i.e., the policy authority) usually has the power to invent and use arbitrary predicates, and to freely decide which clauses are visible.

Since p_{Hi} is freshly chosen, we can safely add it as an atomic clause to the witness candidate without making a difference in terms of observational equivalence. Then instead of adding a clause $P \leftarrow \vec{P}$ that may violate the alikeness requirement, we could have added the equivalent, but invisible, $P \leftarrow p_{\mathrm{Hi}}, \vec{P}$. Lemma 5.7 formalizes this intuition.

Lemma 5.7. Let $A \subseteq \mathbf{Cls}$ such that p_{Hi} does not occur in A. Then there exists $\hat{A} \subseteq \mathbf{Cls}$ such that $\hat{A} \doteq A \cup \{p_{\mathrm{Hi}}\}$ and $\mathbf{Visible}_\ell(\hat{A}) = \emptyset$.

Proof. Let

$$\hat{A} = \{(P \leftarrow p_{\mathrm{Hi}}, \vec{P}) \mid (P \leftarrow \vec{P}) \in A\} \cup \{p_{\mathrm{Hi}}\}.$$

Consider any atom P and set of atoms $\vec{P}$, and let $A_1 = \{P \leftarrow p_{\mathrm{Hi}}, \vec{P}\} \cup \{p_{\mathrm{Hi}}\}$ and $A_2 = \{P \leftarrow \vec{P}\} \cup \{p_{\mathrm{Hi}}\}$. A simple induction on n shows that for all sets S of ground atoms, $\mathbf{T}^n_{A_1 \cup S}(\emptyset) = \mathbf{T}^n_{A_2 \cup S}(\emptyset)$, and hence $A_1 \doteq A_2$. Therefore, $\hat{A} \doteq A \cup \{p_{\mathrm{Hi}}\}$

□

5.5 Initial states

The algorithm will later be presented as a nondeterministic state transition system. A state is a triple $\langle \Pi^+, \Pi^-, A \rangle$, where Π^+ and Π^- are sets of ground probes, and A is a policy. Informally, Π^+ contains the set of probes that have not yet been considered and still need to be made positive in the constructed witness candidate; Π^- contains the set of probes that should be negative in the witness candidate; and A is the current witness candidate. Each transition step eliminates one probe from Π^+ and adds clauses resulting from that probe to A; Π^- stays untouched. A *final state* is therefore of the form $\langle \emptyset, \Pi^-, A'_0 \rangle$. If the final state is produced by a series of transitions starting from an initial state (to be defined in the following), the policy A'_0 (after making it alike to A_0 using the described p_{Hi} method) is guaranteed to be a witness for the opacity of π_0 in A_0, and such a final state exists iff π_0 is opaque in A_0.

Definition 5.8 (Initial state)**.** The rule (INIT) in Fig. 1 defines a set $\mathbf{Init}(\Pi_0)$ of states, parameterized by a set of probes Π_0. We write **Init** to denote $\mathbf{Init}(\mathbf{Avail})$, the set of *initial states.*

$$\text{(INIT)}\ \frac{\begin{array}{c}(\Pi^+,\Pi^-)\in \mathbf{flatten}_{A_0}(\Pi_0)\qquad \pi_0=\langle A,\varphi\rangle\qquad (S^+,S^-)\in \mathbf{dnf}(\neg\varphi)\\ \forall \pi\in\Pi^-\cup\{\langle A,\bigvee S^-\rangle\}:\mathbf{Visible}(A_0)\nvdash\pi\end{array}}{\langle \Pi^+\cup\{\langle A,\bigwedge S^+\rangle\},\ \Pi^-\cup\{\langle A,\bigvee S^-\rangle\},\ \mathbf{Visible}(A_0)\rangle\in\mathbf{Init}(\Pi_0)}$$

$$\text{(PROBE)}\ \frac{\begin{array}{c}\tilde{A}\subseteq A\qquad \langle a_1,...,a_n\rangle\in\mathbf{perms}(\tilde{A})\qquad \forall i\in\{1,...,n\}:\vec{P}_i=\mathbf{bd}(a_i)\qquad \vec{P}_{n+1}=\vec{P}\\ A''=\bigcup_{k=1}^{n+1}\bigcup_{P_k\in\vec{P}_k}\{P_k\leftarrow\mathbf{hds}(\{a_1,...,a_{k-1}\})\}\qquad \forall\pi\in\Pi^-:A'\cup A''\nvdash\pi\end{array}}{\langle \Pi^+\cup\{\langle A,\bigwedge\vec{P}\rangle\},\ \Pi^-,\ A'\rangle\xrightarrow{\langle A,\bigwedge\vec{P}\rangle}\langle\Pi^+,\ \Pi^-,\ A'\cup A''\rangle}$$

Figure 1: Transition system for verifying opacity.

The rule (INIT) in Fig. 1 and Lemma 5.9 encapsulate the observations made so far. The goal of the state transition system is then to find an A_0' as specified in the lemma, starting from the initial states.

Lemma 5.9 (Soundness and completeness of (INIT))**.** The following two statements are equivalent:

1. π_0 is opaque in A_0.
2. There exist $\langle\Pi^+,\Pi^-,\mathbf{Visible}(A_0)\rangle\in\mathbf{Init}$ and $A_0'\subseteq\mathbf{Cls}$ such that p_{Hi} does not occur in A_0' and $\mathbf{Visible}(A_0)\preceq A_0'$ and
$$\forall\pi\in\Pi^+:A_0'\vdash\pi\text{ and }\ \forall\pi\in\Pi^-:A_0'\nvdash\pi.$$

Proof. (2⇒1) If $A_0\nvdash\pi_0$, then 1) is trivially true. Now assume $A_0\vdash\pi_0$. Then by (INIT) and Lemmas 5.5 and 5.3,

$$\forall\pi\in\mathbf{Avail}:A_0'\vdash\pi\iff A_0\vdash\pi\text{ and}\\ A_0'\vdash\neg\pi_0.$$

Since p_{Hi} occurs neither in **Avail** nor in $\neg\pi_0$, there exists $\hat{A}_0\subseteq\mathbf{Cls}$ such that, by Lemma 5.7, $\hat{A}_0\doteq A_0'\cup p_{\mathrm{Hi}}$, and, furthermore, $\mathbf{Visible}(\hat{A}_0)=\emptyset$.

Now consider $\hat{A}_0'=\hat{A}_0\cup\mathbf{Visible}(A_0)$. Clearly, $\hat{A}_0'\simeq A_0$. By (INIT), $A=\mathbf{Visible}(\hat{A}_0)$, so by assumption, $\mathbf{Visible}(A_0)\preceq A_0'\preceq\hat{A}_0$. Therefore, $\hat{A}_0'\doteq\hat{A}_0$.

Since p_{Hi} occurs neither in A_0' nor in π_0 nor in **Avail**, we have

$$\forall\pi\in\mathbf{Avail}:\hat{A}_0'\vdash\pi\iff A_0\vdash\pi\text{ and}\\ \hat{A}_0'\vdash\neg\pi_0.$$

Hence $\hat{A}_0'\equiv A_0$ and $\hat{A}_0'\nvdash\pi_0$, as required.

(1⇒2) If $A_0\nvdash\pi_0$ then the statement trivially holds for $A_0'=A_0$. Now assume $A_0\vdash\pi_0$, and let $\pi_0=\langle A_\pi,\varphi_\pi\rangle$. Then there exists $A_0'\subseteq\mathbf{Cls}$ such that p_{Hi} does not occur in A_0' and $\mathbf{Visible}(A_0)\preceq A_0'$ and

$$\forall\pi\in\mathbf{Avail}:A_0'\vdash\pi\iff A_0\vdash\pi\text{ and}\\ A_0'\vdash\neg\pi_0.$$

Then by the definition of opacity and Lemma 5.5,

$$\exists(\Pi^+, \Pi^-) \in \mathbf{flatten}_{A_0}(\mathbf{Avail}) : \\ (\forall \pi \in \Pi^+ : \ A'_0 \vdash \pi) \text{ and } (\forall \pi \in \Pi^- : \ A'_0 \nvdash \pi).$$

Furthermore, by Lemma 5.3, there exists $(S^+, S^-) \in \mathbf{dnf}(\neg\varphi_\pi)$ such that $A'_0 \vdash \langle A_\pi, \bigwedge S^+ \rangle$ and $A'_0 \nvdash \langle A_\pi, \bigvee S^- \rangle$.

By (INIT), $\langle \Pi^+ \cup \{\langle A_\pi, \bigwedge S^+ \rangle\}, \Pi^- \cup \{\langle A_\pi, \bigvee S^- \rangle\}, \mathbf{Visible}(A_0) \rangle \in \mathbf{Init}$, which completes 2). □

5.6 Minimal witnesses

We want to ensure that the algorithm is a decision procedure, in other words, that it is both complete and terminating, which is necessary for proving that π_0 is *not* opaque (i.e., detectable). However, for every $\langle \Pi^+, \Pi^-, \mathbf{Visible}(A_0) \rangle \in \mathbf{Init}$, there generally exist infinitely many $A'_0 \succeq \mathbf{Visible}(A_0)$ that make all $\pi \in \Pi^+$ positive. There may even be infinitely many A'_0 that additionally also make all $\pi \in \Pi^-$ negative, and are thus genuine witnesses. It is therefore far from obvious that a decision procedure for opacity exists.

Fortunately, it turns out that we do not need to consider all witness candidates A'_0 that make Π^+ positive. Instead, we only compute a particular *finite* subset of those candidates, and prove that it contains all *minimal* witness candidates that make Π^+ positive. These candidates are minimal in the sense that, for every policy A'_0 that makes Π^+ positive, there exists a policy A''_0 with the same property in the computed subset such that $A''_0 \preceq A'_0$.

By antimonotonicity of $\nvdash$, if A'_0 additionally makes Π^- negative, then A''_0 also makes Π^- negative. Hence for every genuine witness there exists a witness in the computed subset that contains it. This property is the basis for our completeness and termination results.

5.7 Relevant subprobe

In order to compute the minimal witnesses, the transition system considers one probe in Π^+ in each transition, and adds 1 or more clauses to the current witness candidate. Consider a state $\langle \Pi^+ \cup \{\pi\}, \Pi^-, A' \rangle$. We have to find *all* minimal ways of extending A' to some policy $A' \cup A''$ such that $A' \cup A'' \vdash \pi$. It turns out that we can ignore A' and simply find all minimal A'' such that $A'' \vdash \pi$; since all $\pi \in \Pi^+$ are monotonic by construction (see (INIT)), it then follows that $A \cup A'' \vdash \pi$.

But how do we construct A''? To gain an intuition for this process, it helps to assume $A'' \vdash \pi$, and then ask the question "why is $\pi = \langle A, \varphi \rangle$ positive in A''?" If A is nonempty, there are multiple explanations. Perhaps φ is true in A'' anyway, so none of the clauses in A are necessary, or relevant, for making π positive. Or perhaps all clauses in A are relevant, in that removing just one clause from A would result in a negative probe. The most general explanation would be that there exists some subset $\tilde{A} \subseteq A$ that is relevant. We need to consider all of these $2^{|A|}$ possible cases, since, as we shall see, each different $\tilde{A}$ results in a different witness extension A'' (and in particular, these resulting A'' are incomparable by the $\preceq$ relation).

Therefore, the transition system is nondeterministic: from the state $\langle \Pi^+ \cup \{\pi\}, \Pi^-, A' \rangle$, there exist multiple states of the form $\langle \Pi^+, \Pi^-, A' \cup A'' \rangle$, reachable in one step, if π has multiple subprobes. This source of branching is reflected in the condition $\tilde{A} \subseteq A$ in the transition rule (PROBE) in Fig. 1.

5.8 Computing witness extensions

Suppose we are considering the probe $\pi = \langle A, \bigwedge \vec{P} \rangle \in \Pi^+$ (this form of the query is dictated by (INIT)). What is the minimal set of clauses A'' that makes π positive, under the assumption that a particular $\tilde{A} \subseteq A$ is relevant?

Since $\tilde{A}$ is relevant, every clause $P_0 \leftarrow P_1, .., P_n \in \tilde{A}$ is actively used at least once in the derivation $A'' \cup \tilde{A} \vdash \bigwedge \vec{P}$. But this is only possible if (i) the body atoms are also derivable, and (ii) the derivation of $\bigwedge \vec{P}$ depends on all the heads of clauses in $\tilde{A}$, i.e., $\mathbf{hds}(\tilde{A})$. We now attempt to solve this set of constraints for the unknown A''.

At first sight, a plausible requirement on A'' seems to be that (i) $\{P_1, ..., P_n\} \subseteq A''$, and (ii) $\{P \leftarrow \mathbf{hds}(\tilde{A}) \mid P \in \vec{P}\} \subseteq A''$. However, while this is a correct solution for A'', it is not the only one. In general, A'' contains the body atoms of a subset of $\tilde{A}$'s clauses, and the heads belonging to these clauses combine with clauses in A'' to make the body atoms of other clauses in $\tilde{A}$ true; this oscillatory back and forth between A'' and $\tilde{A}$ continues until the query $\bigwedge \vec{P}$ is true. The simple solution above corresponds to the special case where the "oscillation" only has one stage.

This process is best illustrated by an example.

Example 5.10. Suppose $\vec{P}$ only contains the nullary atom z, and $\tilde{A}$ contains three clauses:

$$\tilde{A} = \{p \leftarrow q.,\ r \leftarrow s.,\ u \leftarrow v.\}$$

We have to find all minimal A'' such that $A'' \cup \tilde{A} \vdash z$. In the case where the number of stages n is just 1, there is only one minimal solution for A'', containing four clauses:

$$A''_1 = \{q.,\ s.,\ v.,\ z \leftarrow p, r, u.\}$$

In the case $n = 2$, there are six solutions, each containing four clauses; three in which A'' contains one of $\tilde{A}$'s body atoms, and three in which it contains two. Here are two of the six solutions:

$$A''_2 = \{q.,\ s \leftarrow p.,\ v \leftarrow p., z \leftarrow p, r, u.\}$$
$$A''_5 = \{q.,\ s.,\ v \leftarrow p, r.,\ z \leftarrow p, r, u.\}$$

For $n = 3$, there are again six solutions, one for each permutation of $\tilde{A}$, resulting in $1 + 6 + 6 = 13$ solutions:

$$A''_8 = \{q.,\ s \leftarrow p.,\ v \leftarrow p, r.,\ z \leftarrow p, r, u.\}$$
$$A''_9 = \{q.,\ v \leftarrow p.,\ s \leftarrow p, u.,\ z \leftarrow p, r, u.\}$$
$$A''_{10} = \{s.,\ q \leftarrow r.,\ v \leftarrow p, r.,\ z \leftarrow p, r, u.\}$$
$$A''_{11} = \{s.,\ v \leftarrow r.,\ q \leftarrow r, u.,\ z \leftarrow p, r, u.\}$$
$$A''_{12} = \{v.,\ q \leftarrow u.,\ s \leftarrow p, u.,\ z \leftarrow p, r, u.\}$$
$$A''_{13} = \{v.,\ s \leftarrow u.,\ q \leftarrow r, u.,\ z \leftarrow p, r, u.\}$$

But note that A''_8 is contained in ($\preceq$) A''_1, A''_2, and A''_5. Indeed, for each solution from $\{A''_1,...,A''_7\}$, there exists a solution from $\{A''_8, ..., A''_{13}\}$ such that the latter is contained in the former. Hence only the solutions for the case $n = 3$ are minimal. The other solutions need not be considered, since if one of them makes all probes in Π^- negative (and hence is a true witness), then the smaller solution from the case $n = 3$ will also be a true witness, by antimonotonicity of $\nvdash$.

It turns out that this observation holds in the general case. Given a particular $\tilde{A} \subseteq A$, we only need to consider the case $n = |\tilde{A}|$, which has $n!$ solutions. Let **perms** denote the function that maps a set S to the set of all permutations of S. Each permutation of clauses $\langle a_1, ..., a_n\rangle \in \mathbf{perms}(\tilde{A})$ gives rise to a unique witness candidate A'', constructed as in the example above. Let $\vec{P}_i = \mathbf{bd}(a_i)$, for $i \in \{1, ..., n\}$. Then for each $P_1 \in \vec{P}_1$, A'' contains P_1. For each $P_2 \in \vec{P}_2$, it contains the clause $P_2 \leftarrow \mathbf{hd}(a_1)$. For each $P_3 \in \vec{P}_3$, it contains the clause $P_3 \leftarrow \mathbf{hd}(a_1), \mathbf{hd}(a_2)$. In general, for each $P_k \in \vec{P}_k$, A'' contains the clause $P_k \leftarrow \mathbf{hds}(\{a_1, ..., a_{k-1}\})$. Finally, letting $\vec{P}_{n+1} = \vec{P}$, we have that for each $P_{n+1} \in \vec{P}_{n+1}$, A'' contains $P_{n+1} \leftarrow \mathbf{hds}(\{a_1, ..., a_n\})$.

The transition rule (PROBE) in Fig. 1 shows that nondeterministic branching is not only due to picking $\tilde{A} \subseteq A$, but also to picking a permutation from $\mathbf{perms}(\tilde{A})$. The rule constructs A'' as described, and then tests if the new candidate makes all probes in Π^- negative. If it does, then the state transition $\langle \Pi^+ \cup \{\pi\}, \Pi^-, A'\rangle \xrightarrow{\pi} \langle \Pi^+, \Pi^-, A' \cup A''\rangle$ is valid.

The following lemma states the main soundness property of the transition rule.

Lemma 5.11 (Soundness of $\xrightarrow{\pi}$)**.** If $\langle \Pi^+ \cup \{\pi\},\ \Pi^-,\ A'\rangle \xrightarrow{\pi} \langle \Pi^+,\ \Pi^-,\ A' \cup A''\rangle$, then $A'' \vdash \pi$.

Proof. By (PROBE), π must be of the form $\langle A, \bigwedge \vec{P}\rangle$. We will prove the following, stronger, statement.

For all $\tilde{A} \subseteq A$, $\langle a_1, \ldots, a_n\rangle \in \mathbf{perms}(\tilde{A})$ and

$$A''_0 = \bigcup_{k=1}^{n} \bigcup_{P_k \in \mathbf{bd}(a_k)} \{P_k \leftarrow \mathbf{hds}(\{a_1, ..., a_{k-1}\})\} \tag{13}$$

we have $A''_0 \cup \tilde{A} \vdash \bigwedge \mathbf{hds}(\tilde{A})$.

This implies the statement of the lemma, since, by the definition of (PROBE), $A'' = A''_0 \cup \{P \leftarrow \mathbf{hds}(\tilde{A})\}$ is precisely the set of assertions added to the state in a $\xrightarrow{\pi}$ transition; furthermore, for all $P \in \vec{P}$: $(P \leftarrow \mathbf{hds}(\tilde{A})) \in A''$ and $\tilde{A} \subseteq A$, and therefore:

$$A'' \cup \tilde{A} \vdash \bigwedge \mathbf{hds}(\tilde{A}) \Rightarrow A'' \cup A \vdash \bigwedge \vec{P} \Rightarrow A'' \vdash \pi. \tag{14}$$

The proof proceeds by induction on n. In the base case, $\tilde{A} = \mathbf{hds}(\tilde{A}) = \emptyset$. Then $\mathbf{hds}(\tilde{A}) = \emptyset$, so the statement trivially holds.

In the inductive case, $n = |\tilde{A}| > 0$. Consider any $\langle a_1, \ldots, a_n\rangle \in \mathbf{perms}(\tilde{A})$. Let A''_0

be defined as in (13). Then we have $A_0'' = A_1'' \cup A_2''$, where

$$A_1'' = \bigcup_{k=1}^{n-1} \bigcup_{P_k \in \mathbf{bd}(a_k)} \{P_k \leftarrow \mathbf{hds}(\{a_1, ..., a_{k-1}\})\},$$
$$A_2'' = \bigcup_{P_k \in \mathbf{bd}(a_n)} \{P_k \leftarrow \mathbf{hds}(\{a_1, ..., a_{n-1}\})\}.$$

By the induction hypothesis, $A_1'' \cup \{a_1, \ldots, a_{n-1}\} \vdash \bigwedge \mathbf{hds}(\{a_1, \ldots, a_{n-1}\})$. By monotonicity of $\vdash$, it also holds that

$$A_0'' \cup \tilde{A} \vdash \bigwedge \mathbf{hds}(\{a_1, \ldots, a_{n-1}\}). \tag{15}$$

It remains to show that $A_0'' \cup \tilde{A} \vdash \mathbf{hd}(a_n)$.

By the definition of A_2'' and by (15), and since $A_2'' \subseteq A_0''$, we have $A_0'' \cup \tilde{A} \vdash \bigwedge \mathbf{bd}(a_n)$. Since $a_n = (\mathbf{hd}(a_n) \leftarrow \mathbf{bd}(a_n)) \in \tilde{A}$, this implies $A_0'' \cup \tilde{A}' \vdash \mathbf{hd}(a_n)$, as required. □

The main completeness property of the transition rule is a bit harder to state. In the following lemma, we consider a policy A_1 and a probe $\langle A_2, \bigwedge \vec{P} \rangle$ that is positive in A_1. Roughly speaking, A_1 in the lemma plays the role of A'' in the transition rule, and A_2 that of $\tilde{A}$. The clause set A_2 is relevant in the sense that every proper subprobe is negative in A_1. We then prove that the clauses constructed by the transition rule must be contained (in the $\preceq$ sense) in A_1.

Lemma 5.12 (Completeness of $\xrightarrow{\pi}$)**.** Let A_1, A_2 be policies with $n = |A_2|$, and $\vec{P}$ a set of ground atoms. If $A_1 \vdash \langle A_2, \bigwedge \vec{P} \rangle$ and for all $A_2' \subsetneq A_2 : A_1 \nvdash \langle A_2', \bigwedge \vec{P} \rangle$, then there exist $\langle a_1, ..., a_n \rangle \in \mathbf{perms}(A_2)$ and

$$\begin{aligned} \vec{P}_i &= \mathbf{bd}(a_i), \text{ for } i \in \{1, ..., n\}, \text{ and} \\ \vec{P}_{n+1} &= \vec{P} \end{aligned}$$

such that for all $i \in \{1, ..., n+1\}$:

$$\forall P_i \in \vec{P}_i : (P_i \leftarrow \mathbf{hds}(\{a_1, ..., a_{i-1}\})) \preceq A_1.$$

Proof. Let $\Phi(A_1, A_2, \vec{P})$ denote the parameterized statement of the lemma. The proof proceeds by induction on $n = |A_2|$. If $n = 0$, the statement trivially holds.

Now assume $n > 0$. By assumption, A_2 is minimal. Therefore, there exists a smallest integer m such that $\mathbf{hds}(A_2) \subseteq \mathbf{T}^m_{A_1 \cup A_2}(\emptyset)$. Furthermore, $m \geq 1$, since $n > 0$. Hence there exists $\tilde{A}_2$, the largest subset of A_2 such that $\mathbf{hds}(\tilde{A}_2) \cap \mathbf{T}^{m-1}_{A_1 \cup A_2}(\emptyset) = \emptyset$, and such that $k = |\tilde{A}_2| \geq 1$.

Let $\vec{P}'$ be the set of all body atoms of clauses in $\tilde{A}_2$. By construction of $\tilde{A}_2$ and since $\mathbf{hds}(\tilde{A}_2) \subseteq \mathbf{hds}(A_2) \subseteq \mathbf{T}^m_{A_1 \cup A_2}(\emptyset)$, we have that $\vec{P}' \subseteq \mathbf{T}^{m-1}_{A_1 \cup A_2}(\emptyset)$, and again by construction of $\tilde{A}_2$, we also have $\vec{P}' \subseteq \mathbf{T}^{m-1}_{A_1 \cup (A_2 \setminus \tilde{A}_2)}(\emptyset)$. Since $|A_2 \setminus \tilde{A}_2| = n - k < n$, we can assume the inductive hypothesis $\Phi(A_1, A_2 \setminus \tilde{A}_2, \vec{P}')$, and in particular, the existence of $\langle a_1, ..., a_{n-k} \rangle \in \mathbf{perms}(A_2 \setminus \tilde{A}_2)$ with the stated properties.

Let $\langle a_{n-k+1}, ..., a_n \rangle$ be any permutation of $\tilde{A}_2$. We thus have constructed a permutation $\langle a_1, ..., a_n \rangle \in \mathbf{perms}(A_2)$. This permutation gives rise to sets $S_0, ..., S_n$ and $\vec{P}_1, ..., \vec{P}_{n+1}$ as defined in the lemma.

For $i \in \{1, ..., n-k\}$, the desired property follows directly from the inductive hypothesis. Furthermore, the inductive hypothesis states that

$$\forall P' \in \vec{P}' : (P' \leftarrow \mathbf{hds}(\{a_1, ..., a_{n-k}\})) \preceq A_1.$$

Then for $i \in \{n-k+1, ..., n\}$, we get (since $\vec{P}_i \subseteq \vec{P}'$):

$$\begin{array}{ll} \forall P_i \in \vec{P}_i : & (P_i \leftarrow \mathbf{hds}(\{a_1, ..., a_{i-1}\})) \preceq \\ & (P_i \leftarrow \mathbf{hds}(\{a_1, ..., a_{n-k}\})) \preceq A_1. \end{array}$$

It thus remains to consider the case $i = n+1$. From $A_1 \vdash \langle A_2, \bigwedge \vec{P}_{n+1} \rangle$ and $A_2 \preceq \mathbf{hds}(A_2)$ we get $\forall P_{n+1} \in \vec{P}_{n+1} : A_1 \vdash \langle \mathbf{hds}(A_2), P_{n+1} \rangle$, and hence by Lemma 3.8,

$$(P_{n+1} \leftarrow \mathbf{hds}(\{a_1, ..., a_n\})) \preceq A_1,$$

as required. □

5.9 A full example

Consider the following scenario, where A denotes the clause set $\{p \leftarrow q.,\ r \leftarrow s.,\ u \leftarrow v.\}$.

$$\begin{array}{l} \mathbf{Avail} = \{\langle A', z \rangle \mid A' \subseteq A\} \\ A_0 = \{q.,\ s.,\ v.,\ z \leftarrow p, r, u.\} \\ \mathbf{Visible}(A_0) = \emptyset \\ \pi_0 = \langle \emptyset, q \vee s \rangle \end{array}$$

We are interested in the question if π_0 is opaque or detectable in A_0.

It is easy to see that $\pi = \langle A, z \rangle$ is the only probe in **Avail** that is positive in A_0; for all other probes $\pi' \in \mathbf{Avail}$, $A_0 \nvdash \pi'$. Since the queries in the available probes do not involve disjunctions (they are all equal to z), $\mathbf{flatten}_{A_0}(\mathbf{Avail})$ contains only one pair of probe sets $(\{\pi\}, \mathbf{Avail} \setminus \{\pi\})$. Furthermore, $\mathbf{dnf}(\neg(q \vee s))$ also contains only one pair of atom sets $(S^+, S^-) = (\emptyset, \{q, s\})$. Hence, **Init** contains only one initial state $\sigma_0 = \langle \Pi^+, \Pi^-, \emptyset \rangle$, where $\Pi^+ = \{\pi\}$ and $\Pi^- = \mathbf{Avail} \setminus \{\pi\} \cup \{\langle \emptyset, q \vee s \rangle\}$.

Starting from σ_0, π is the only probe to be considered, so the transition system tries out all $\tilde{A} \subseteq A$. For all $\tilde{A} \subsetneq A$, no transition is possible, as all resulting witness candidates fail to make Π^- negative, because $\langle \tilde{A}, z \rangle \in \Pi^-$. For $\tilde{A} = A$, the transition system goes through all six permutations of $\tilde{A}$, and produces the six witness candidates A''_8–A''_{13} from Example 5.10. The first four of these clearly fail to make $\langle \emptyset, q \vee s \rangle \in \Pi^-$ negative. But A''_{12} (and also A''_{13}) passes the test, so $\langle \emptyset, \Pi^-, A''_{12} \rangle$ is a final state, and hence π_0 is opaque.

We can easily produce a genuine opacity witness from A''_{12} by adding p_{Hi} to it and injecting p_{Hi} into the bodies of all clauses in A''_{12}. The resulting policy is observationally equivalent to A_0, but $q \vee s$ does not hold in it.

5.10 Correctness results

We can now state and prove soundness and completeness of the algorithm. The proof of Theorem 5.16 makes essential use of all the previous lemmas.

Definition 5.13 (Reachability)**.** We write $\sigma \to \sigma'$ to denote that $\sigma \xrightarrow{\pi} \sigma'$ for some $\pi \in \mathbf{Prb}$ and states σ, σ'. We write $\to^*$ for the reflexive-transitive closure of $\to$. We write $\vdash \sigma$ if $\sigma_0 \to^* \sigma$ for some $\sigma_0 \in \mathbf{Init}$.

Lemma 5.14 (Soundness)**.** If $\vdash \langle \emptyset, \Pi_0^-, A \rangle$, then π_0 is opaque in A_0.

Proof. If $\vdash \langle \emptyset, \Pi_0^-, A \rangle$, their exists $\sigma_I = \langle \Pi^+, \Pi^-, \mathbf{Visible}(A_0) \rangle \in \mathbf{Init}$ such that $\sigma_I \to^* \langle \emptyset, \Pi_0^-, A \rangle$, where Π^+, Π^- are specified as in (INIT), and $\Pi^- = \Pi_0^-$.

There is a series of (PROBE) applications starting from σ_I, with one $\xrightarrow{\pi}$ transition for each $\pi \in \Pi^+$, leading to $\langle \emptyset, \Pi_0^-, A \rangle$. Hence by repeated application of Lemma 5.11, we have $\forall \pi \in \Pi^+ : A \vdash \pi$. From the definitions of (INIT) and (PROBE) it follows that $\forall \pi \in \Pi^- : A \nvdash \pi$, and that p_{Hi} does not occur in A. Furthermore, $\mathbf{Visible}(A_0) \subseteq A$. Therefore, by Lemma 5.9, π_0 is opaque in A_0. □

Lemma 5.15 (Completeness)**.** If π_0 is opaque in A_0, then there exists $\Pi_0^- \subseteq \mathbf{Prb}$ and $A \subseteq \mathbf{Cls}$ such that

$$\vdash \langle \emptyset, \Pi_0^-, A \rangle.$$

Proof. If π_0 is opaque in A_0, then by Lemma 5.9, there exists $\sigma_I = \langle \Pi_0^+, \Pi_0^-, A_I \rangle \in \mathbf{Init}$ and $A' \subseteq \mathbf{Cls}$ such that $A_I \preceq A'$ and $\forall \pi \in \Pi^+ : A' \vdash \pi$ and $\forall \pi \in \Pi^- : A' \nvdash \pi$.

By Lemma 5.5, there exists $(\Pi^+, \Pi^-) \in \mathbf{flatten}_{A_0}(\mathbf{Avail})$ such that $\forall \pi \in \Pi^+ : \ A' \vdash \pi$ and $\forall \pi \in \Pi^- : \ A' \nvdash \pi$. Furthermore, by Lemma 5.3, there exists (Π_S^+, Π_S^-) as defined in (INIT) such that $\forall \pi \in \Pi_S^+ : \ A' \vdash \pi$ and $\forall \pi \in \Pi_S^- : \ A' \nvdash \pi$. Let $\Pi_0^+ = \Pi^+ \cup \Pi_S^+$ and $\Pi_0^- = \Pi^- \cup \Pi_S^-$. Then $\sigma_I = \langle \Pi_0^+, \Pi_0^-, \emptyset \rangle \in \mathbf{Init}$.

We now prove that for all $\Pi_1^+ \subseteq \Pi_0^+$, there exists $A \preceq A'$ such that

$$\langle \Pi_1^+, \Pi_0^-, A_I \rangle \to^* \langle \emptyset, \Pi_0^-, A \rangle.$$

The proof proceeds by induction on $m = |\Pi_1^+|$. If $m = 0$, the statement holds trivially.

If $m > 0$, there exists $\pi = \langle A_\pi, \bigwedge \vec{P} \rangle \in \Pi_1^+$. By the inductive hypothesis, there exists $A_1 \preceq A'$ such that $\langle \Pi_1^+ \setminus \{\pi\}, \Pi_0^-, A_I \rangle \to^* \langle \emptyset, \Pi_0^-, A_1 \rangle$. By inspection of (PROBE), we also have $\langle \Pi_1^+, \Pi_0^-, A_I \rangle \to^* \sigma = \langle \{\pi\}, \Pi_0^-, A_1 \rangle$. It remains to show that there exists $A \preceq A'$ such that $\sigma \xrightarrow{\pi} \langle \emptyset, \Pi_0^-, A \rangle$.

Since $\pi \in \Pi_0^+$, we have $A' \vdash \pi$. Then there exists a minimal $\tilde{A}$ such that $\tilde{A} \subseteq A_\pi$ and $A' \vdash \langle \tilde{A}, \bigwedge \vec{P} \rangle$. Let $n = |\tilde{A}|$. Then by Lemma 5.12, there exists $\langle a_1, ..., a_n \rangle \in \mathbf{perms}(A_\pi)$ and there exist

$$\begin{aligned} \vec{P}_i \quad &= \mathbf{bd}(a_i), \text{ for } i \in \{1, ..., n\}, \text{ and} \\ \vec{P}_{n+1} &= \vec{P} \end{aligned}$$

such that for all $k \in \{1, ..., n+1\}$ and all $P_k \in \vec{P}_k$

$$(P_k \leftarrow \mathbf{hds}(\{a_1, ..., a_{k-1}\})) \preceq A'.$$

Hence $A'' = \bigcup_{k=1}^{n+1} \bigcup_{P_k \in \vec{P}_k} \{P_k \leftarrow \mathbf{hds}(\{a_1, ..., a_{k-1}\})\} \preceq A'$, and thus there exists $A = A_1 \cup A'' \preceq A'$. By anti-monotonicity of $\nvdash$, we get $\forall \pi' \in \Pi_0^- : A \nvdash \pi'$. Hence all conditions of (PROBE) are satisfied.

Therefore, $\sigma_i \rightarrow^* \sigma \xrightarrow{\pi} \langle \emptyset, \Pi_0^-, A \rangle$.

□

Theorem 5.16 (Soundness and completeness)**.** π_0 is opaque in A_0 iff there exist $\Pi^- \subseteq \mathbf{Prb}$ and $A \subseteq \mathbf{Cls}$ such that $\vdash \langle \emptyset, \Pi^-, A \rangle$.

Proof. This follows directly from Lemmas 5.14 and 5.15.

□

Theorem 5.17. The number of ($\xrightarrow{\langle A, \bigwedge \vec{P} \rangle}$) transitions from any state is bounded by

$$\sum_{m=0}^{n} \frac{n!}{(n-m)!}$$

where $n = |A|$.

Proof. The number of $\tilde{A} \subseteq A$ of size m is $\binom{n}{m}$, and for each such $\tilde{A}$, there are $m!$ permutations. The size m runs from 0 to m, hence the number of transitions is bounded by

$$\sum_{m=0}^{n} m! \binom{n}{m} = \sum_{m=0}^{n} \frac{n!}{(n-m)!}.$$

□

Theorem 5.17 also implies that the transition system is finite, and hence the algorithm terminates, since **Init** is finite, and Π^+ in every initial state is finite.

6 Implementation with optimizations

We implemented a prototype of the state transition system in Fig. 1 in F#. It first computes **Init** as a lazy enumeration, and then performs a backtracking depth first search based on the transition rule (PROBE). The back end is a simple implementation of Datalog's evaluation relation $\vdash$. Even though it is the main bottleneck, we did not spent much effort making it more efficient, as we are more interested in algorithmic improvements of the search procedure.

The front end includes a parser for problem specifications (A_0, **Visible**(A_0), **Avail**, and π_0) and a GUI that displays the witness, if a final state has been found, or reports that no final state exists. In the former case, the user can choose to discard the found witness and continue the search for the next witness. We have found this to be an extremely useful feature for checking opacity. In some of our test cases, we expected the input probe to be detectable, but the prototype reported opacity. As it turned out, the input probe

was indeed opaque, but all minimal witnesses included "improbable" clauses. Thus, the constructiveness of the algorithm enabled us to form the informal judgement that an input probe was detectable with a rather high likelihood. By inspecting all witnesses, it was also easy to come up with a weakened input probe that was truly detectable. Section 7 describes an example of such a case.

As Theorem 5.17 indicates, the search space is huge, so checking opacity is infeasible even for small examples. Therefore, it was important to come up with and to implement methods for effectively pruning the search space. The remainder of this section describes the pruning methods we implemented.

6.1 Order independence

The most obvious way to prune the search space is based on the observation that the order in which the probes in Π^+ are processed is irrelevant, since the constructed witness extension A'' is independent of the current witness candidate; it only depends on the currently considered probe. Lemma 6.1 formalizes this observation.

Lemma 6.1 (Order independence)**.** If $\sigma_0 \xrightarrow{\pi_1} \sigma_1 \xrightarrow{\pi_2} \sigma_2$ then there exists σ_1' such that $\sigma_0 \xrightarrow{\pi_2} \sigma_1' \xrightarrow{\pi_1} \sigma_2$.

Proof. This follows from the definition of (PROBE). □

Therefore, we can fix a particular order for Π^+ in an initial state, thereby reducing the search space by a factor of $|\Pi^+|!$ for the search branch starting from that initial state.

6.2 Redundant probes

There is often a clause set A such that **Avail** contains all probes of the form $\langle A', \varphi\rangle$, for all $A' \subseteq A$, since if the adversary is able to use the probe $\pi = \langle A, \varphi\rangle$, it is likely that she is also able to use all smaller probes. In our implementation, the user can add the entire *downward closed* probe set $\{\langle A', \varphi\rangle \mid A' \subseteq A\}$ to **Avail** simply by marking the single probe $\langle A, \varphi\rangle$ with a plus sign in the specification of **Avail**.

Definition 6.2. Let $\pi = \langle A, \varphi\rangle, \pi' = \langle A', \varphi'\rangle \in \mathbf{Prb}$. We write $\pi \subseteq \pi'$ iff $\varphi = \varphi'$ and $A \subseteq A'$. Similarly, we write $\pi \subsetneq \pi'$ iff $\pi \subseteq \pi'$ and $\pi \neq \pi'$.

In the presence of downward closed probe sets in **Avail**, Π^+ and Π^- contain many pairs of probes π_1, π_2 such that $\pi_1 \subseteq \pi_2$.

For example, we may have $\pi_1 = \langle\{a.\}, z\rangle$ as well as $\pi_2 = \langle\{a.,\ b.\}, z\rangle$ in Π^+. By monotonicity of $\vdash$ and of the query z, the larger query π_2 is redundant, since any witness candidate that makes π_1 positive also makes π_2 positive. A similar argument can be made for the probes in Π^-. Lemma 6.3 formalizes the notion of redundant probes.

Lemma 6.3. Let π_1, π_2 be monotonic probes such that $\pi_1 \subseteq \pi_2$.

1. $\exists A' : \langle \Pi^+ \cup \{\pi_1, \pi_2\}, \Pi^-, A\rangle \rightarrow^* \langle \emptyset, \Pi^-, A'\rangle$ iff
   $\exists A'' : \langle \Pi^+ \cup \{\pi_1\}, \Pi^-, A\rangle \rightarrow^* \langle \emptyset, \Pi^-, A''\rangle$ and $A'' \vdash \pi_2$.

2. $\exists A' : \langle \Pi^+, \Pi^- \cup \{\pi_1, \pi_2\}, A\rangle \rightarrow^* \langle \emptyset, \Pi^- \cup \{\pi_1, \pi_2\}, A'\rangle$ iff $\exists A'' :\ \langle \Pi^+, \Pi^- \cup \{\pi_2\}, A\rangle \rightarrow^* \langle \emptyset, \Pi^- \cup \{\pi_2\}, A''\rangle$ and $A'' \nvdash \pi_1$.

Proof. The lemma is trivially true if $\pi_1 = \pi_2$. We now assume $\pi_1 \subsetneq \pi_2$.

1. ($\Rightarrow$) From the definition of (PROBE) and the left hand side of the equivalence, the existence of A'', such that the first part of the right hand side holds, is clear. By Lemma 5.11, we have $A'' \vdash \pi_1$, which, by monotonicity of $\vdash$ and π_1 and π_2, implies $A'' \vdash \pi_2$.

   ($\Leftarrow$) The right hand side of the equivalence implies $\langle \Pi^+ \cup \{\pi_1, \pi_2\}, \Pi^-, A\rangle \rightarrow^* \langle \{\pi_2\}, \Pi^-, A''\rangle$, and in particular, within this chain there exists a transition $\sigma \xrightarrow{\pi_1, \vec{a}} \sigma'$ for some states $\sigma, \sigma', \vec{a}$. Since $\pi_1 \subsetneq \pi_2$, we have $\langle \{\pi_2\}, \Pi^-, A''\rangle \xrightarrow{\pi_2, \vec{a}} \langle \emptyset, \Pi^-, A'\rangle$, where $A' = A''$.

2. ($\Rightarrow$) Assuming the left hand side of the equivalence, the existence of A'', such that the first part of the right hand side holds, is clear. From the definition of (PROBE), we have $A'' \nvdash \pi_2$, which, by antimonotonicity of $\nvdash$ and the monotonocity of π_1 and π_2, implies $A'' \nvdash \pi_1$.

   ($\Rightarrow$) The right hand side of the equivalence implies that $A'' \nvdash \pi_1$ and $A'' \nvdash \pi_2$, and hence $\langle \Pi^+, \Pi^- \cup \{\pi_1, \pi_2\}, A\rangle \rightarrow^* \langle \emptyset, \Pi^- \cup \{\pi_1, \pi_2\}, A'\rangle$, where $A' = A''$.

□

To exploit Lemma 6.3, we can first transform initial states $\langle \Pi_0^+, \Pi_0^-, A\rangle \in \mathbf{Init}$ into potentially much smaller states $\langle \Pi_1^+, \Pi_1^-, A\rangle$, where

$$\Pi_1^+ = \{\pi \in \Pi_0^+ \mid \neg\exists \pi' \in \Pi_0^+ :\ \pi' \subsetneq \pi\}, \text{ and}$$
$$\Pi_1^- = \{\pi \in \Pi_0^- \mid \neg\exists \pi' \in \Pi_0^- :\ \pi \subsetneq \pi'\}.$$

These reduced states are then used as initial states.

6.3 Conflicting probes

Lemma 6.4 provides an additional way of pruning the search space. Any initial state $\sigma_0 = \langle \Pi^+, \Pi^-, A\rangle$ in which there exist $\pi_1 \in \Pi^+, \pi_2 \in \Pi^-$ such that $\pi_1 \subseteq \pi_2$ can be discarded straight away, as there are no transitions from σ_0.

Lemma 6.4. Let π_1, π_2 be monotonic probes such that $\pi_1 \subseteq \pi_2$. There exists no state σ such that $\langle \Pi^+ \cup \{\pi_1\}, \Pi^- \cup \{\pi_2\}, A\rangle \xrightarrow{\pi_1} \sigma$.

Proof. Suppose the contrary. Then there exists $\sigma = \langle \Pi^+, \Pi^- \cup \{\pi_2\}, A'\rangle$ with the property as stated. Moreover, $A' \vdash \pi_1$, by Lemma 5.11, and $A' \nvdash \pi_2$, by (PROBE). Since $\pi_1 \subseteq \pi_2$, the former implies $A' \vdash \pi_2$, which contradicts the latter.

□

It may not be obvious that Lemma 6.4 ever applies to initial states in **Init**. Here is a simple example. Consider $\pi_1 = \langle A_1, p \wedge \neg q\rangle$ and $\pi_2 = \langle A_2, p \wedge \neg q\rangle \subsetneq \pi_1$. Suppose that $A_0 \vdash \pi_1$ and $A_0 \nvdash \pi_2$. If $\mathbf{Avail} = \{\pi_1, \pi_2\}$ then $\sigma_0 = \langle \{\langle A_1, p\rangle, \langle A_2, q\rangle\}, \{\langle A_1, q\rangle\}, \mathbf{Visible}(A_0)\rangle \in \mathbf{Init}$, and clearly $\langle A_2, q\rangle \subseteq \langle A_1, q\rangle$, so σ_0 should be discarded.

6.4 Minimally positive probes

Consider an initial state $\sigma_0 = \langle \Pi^+, \Pi^-, \mathbf{Visible}(A_0) \rangle$ where Π^+ contains a probe $\pi^+ = \langle \{a_1, ..., a_n\}, \varphi \rangle$. Let $A = \{a_1, ..., a_n\}$. Suppose further that for each $i \in \{1, ..., n\}$, Π^- contains a probe π_i^- such that $\pi_i^- \supseteq \langle A \setminus \{a_i\}, \varphi \rangle$. It follows that π^+ is *minimally positive* in A_0 in the sense that π^+ is positive in A_0 and all probes that are strictly smaller than π^+ are negative in A_0.

Definition 6.5. A probe π is *downward closed* in $\Pi \subseteq \mathbf{Prb}$ iff for all $\pi' \in \mathbf{Prb}$: $\pi' \subseteq \pi \Rightarrow \pi' \in \Pi$.

A probe $\pi = \langle A, \varphi \rangle \in \mathbf{Prb}$ is *minimally positive* in $A' \subseteq \mathbf{Cls}$ iff φ is monotonic and $A' \vdash \pi$ and for all $\pi' \subsetneq \pi : A' \nvdash \pi$.

We can exploit this fact when π^+ is considered in a (PROBE) transition. We first introduce a new notation.

Notation 6.6. In a (PROBE) transition $\sigma \xrightarrow{\pi} \sigma'$, the state σ' is determined by σ, $\pi = \langle A, \bigwedge \vec{P} \rangle$ and $\vec{a} \in \mathbf{perms}(\tilde{A})$ for some $\tilde{A} \subseteq A$. We write $\sigma \xrightarrow{\pi, \vec{a}} \sigma'$ to make these determining parameters explicit.

Suppose σ is a state reachable from σ_0, and $\sigma \xrightarrow{\pi^+, \vec{a}} \sigma'$ for some state σ' and some $\vec{a} \in \mathbf{perms}(\tilde{A})$ where $\tilde{A} \subseteq A$. We argue that, in fact, $\tilde{A} = A$, for suppose that $\tilde{A} \subsetneq A$ were the case. But this would be only possible if $A_0 \vdash \langle \tilde{A}, \varphi \rangle$, which contradicts the fact that π^+ is minimally positive in A_0. Hence $\tilde{A} = A$. This observation is formalized in Lemma 6.7.

Lemma 6.7. Let $\pi = \langle A, \varphi \rangle$ be a monotonic probe, where $A = \{a_1, ..., a_n\}$ for some $n \geq 0$. Let $\pi_i \in \mathbf{Prb}$ with $\pi_i \supseteq \langle A \setminus \{a_i\}, \varphi \rangle$, for $i \in \{1, ..., n\}$. If $\langle \Pi^+ \cup \{\pi\}, \Pi^-, A' \rangle \xrightarrow{\pi, \vec{a}} \langle \Pi^+, \Pi^-, A'' \rangle$ and $\{\pi_1, ..., \pi_n\} \subseteq \Pi^-$, then $\vec{a} \in \mathbf{perms}(A)$.

Proof. Suppose for the sake of contradiction that $\vec{a} \in \mathbf{perms}(\tilde{A})$ where $\tilde{A} \subsetneq A$.

Note that Lemma 5.11 can be strengthened, with only slight modification of the proof, to the following statement: if $\langle \Pi^+ \cup \{\langle A, \varphi \rangle\}, \ \Pi^-, \ A' \rangle \xrightarrow{\langle A, \varphi \rangle, \vec{a}} \langle \Pi^+, \ \Pi^-, \ A'' \rangle$, then $A'' \vdash \langle \vec{a}, \varphi \rangle$.

Hence we have $A'' \vdash \langle \vec{a}, \varphi \rangle$. But since $\tilde{A} \subsetneq A$, there exists $i \in \{1, ..., n\}$ such that $\tilde{A} \subseteq A \setminus \{a_i\}$. Since $\pi_i \in \Pi^-$, we have $A'' \nvdash \pi_i \supseteq \langle A \setminus \{a_i\}, \varphi \rangle$. Since φ is monotonic, we also have $A'' \nvdash \langle \tilde{A}, \varphi \rangle = \langle \vec{a}, \varphi \rangle$, which is a contradiction. Hence $\tilde{A} = A$, and $\vec{a} \in \mathbf{perms}(A)$, as required. □

Therefore, when π^+ is considered, we replace (PROBE) by a much more efficient transition rule where the nondeterministic condition $\tilde{A} \subseteq A$ is replaced by the non-branching $\tilde{A} = A$. This optimization reduces the number of transition attempts (from that particular state) by a factor of $2^{|A|}$.

In our prototype implementation, the user can declare a probe in **Avail** as minimally positive by marking it with an asterisk.

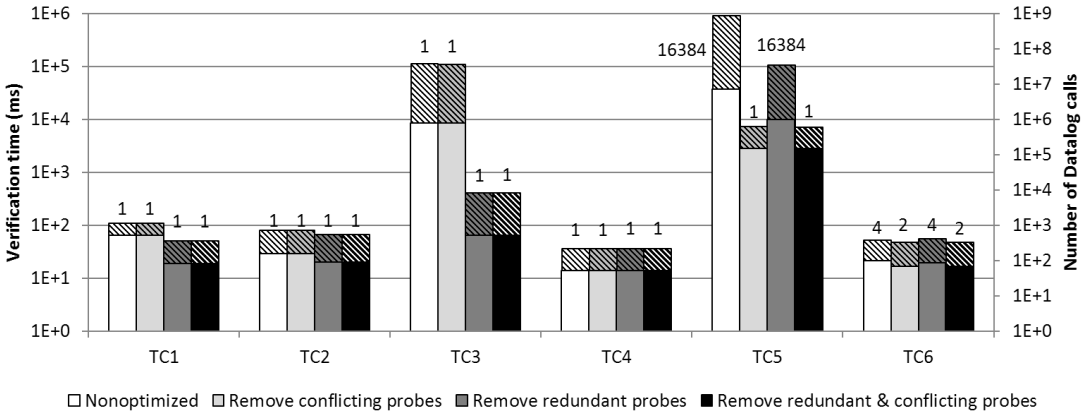

Figure 2: Verification time (striped) and number of calls to the Datalog engine (plain) on log scales. Numbers above bars denote the number of initial states.

7 Experimental results

We are interested in typical computation times of opacity checking in a realistic setting, in the relative effectiveness of the optimization methods described in Section 6, and in how the method scales with varying available probe sets.

7.1 Test cases

Our performance tests are based on the delegation policy from Section 4. From this policy, we derived six test cases (TC1–TC6), described below. For each test case, we measured both the computation times and the number of calls to the Datalog engine for proving opacity or detectability. Each test case was run in four different configurations of the prototype. All four configurations use the order independence optimization from Section 6.1. The first configuration does not employ any other optimization method; the second eliminates conflicting probes (Section 6.3); the third eliminates redundant probes (Section 6.2); and in the fourth configuration, both optimization methods are enabled. The experiments were performed on an Intel Core2 Duo P9500 2.53GHz workstation with 4 GB RAM running Windows 7 32-bit.

Fig. 2 shows a bar diagram of the respective computation times and the number of Datalog calls plotted against logarithmic scales. The diagram also shows the number of initial states for each run.

TC1. As in Section 4, policy A_0 in our basic test case consists of the compute cluster's policy, i.e., clauses (1)–(7), and $\mathbf{Visible}(A_0) = \emptyset$.

The adversary `Eve` possesses four credentials $A_{\texttt{Eve}} = \{(9), ..., (12)\}$. We define

$$\varphi_{\texttt{Eve}} = \mathsf{canExec}(\texttt{Cluster}, \texttt{Eve}, \texttt{Job}).$$

The probe set **Avail** available to `Eve` consists of $2^4 = 16$ probes:

$$\mathbf{Avail} = \{\langle A, \varphi_{\texttt{Eve}}\rangle \mid A \subseteq A_{\texttt{Eve}}\}.$$

In this test case, we are interested in whether `Eve` can detect that `Bob` is not a member of `Cluster`, in other words, that $A_0 \vdash \neg\mathsf{isMem}(\texttt{Cluster}, \texttt{Bob})$ holds. The input probe π_0 is therefore specified as follows:

$$\pi_0 = \langle \emptyset, \neg\mathsf{isMem}(\texttt{Cluster}, \texttt{Bob})\rangle.$$

As was informally shown in Section 4, π_0 is detectable in A_0.

TC2. Based on TC1, the atomic clause $\mathsf{isMem}(\texttt{Cluster}, \texttt{Bob})$ is added to A_0 (so now, `Bob` *is* a member) and the input probe π_0 is changed to $\langle \emptyset, \mathsf{isMem}(\texttt{Cluster}, \texttt{Bob})\rangle$.

At first sight, it seems that `Eve` should be able to detect that `Bob` is a member. After all, the probe containing clauses (9), (10) and (12) is positive, whereas the one containing only (9) and (10) is negative. This suggests that (12) is relevant, which is only possible if its body atom $\mathsf{isMem}(\texttt{Cluster}, \texttt{Bob})$ is derivable.

However, the prototype (correctly) reports that π_0 is opaque. A closer look at the witnesses reveals that they all contain the rather unlikely clause that has (9) and (10) as its body:

$$\mathsf{isMem}(\texttt{Cluster}, \texttt{Bob}) \leftarrow \\ \quad \mathsf{isMem}(\texttt{CA}, \texttt{Eve}), \mathsf{owns}(\texttt{CA}, \texttt{Eve}, \texttt{Job}).$$

Indeed, the weakened input probe

$$\pi_0' = \langle \{(9), (10)\}, \mathsf{isMem}(\texttt{Cluster}, \texttt{Bob})\rangle$$

is detectable.

TC3. Based on TC1, three irrelevant atomic clauses $\{p_1., \ p_2., \ p_3.\}$ are added to $A_{\texttt{Eve}}$. This increases the number of probes in **Avail** to $2^7 = 128$, i.e., by a factor of 8.

TC4. Based on TC3, we manually prune **Avail** down to four probes that together are sufficient for proving detectability of π_0:

$$\langle \{(9), (10), (11)\}, \varphi_{\texttt{Eve}}\rangle^* \\ \langle \{(9), (10), (12), \ p_1., \ p_2., \ p_3.\}, \varphi_{\texttt{Eve}}\rangle \\ \langle \{(10), (11), (12), \ p_1., \ p_2., \ p_3.\}, \varphi_{\texttt{Eve}}\rangle \\ \langle \{(9), (11), (12), \ p_1., \ p_2., \ p_3.\}, \varphi_{\texttt{Eve}}\rangle$$

Only the first of these four probes is positive. The four probes together satisfy the conditions of Lemma 6.7, so we mark the first probe as minimally positive (*cf.* Section 6.4). This enables the use of the much more efficient version of (PROBE) that only needs to consider the case $\tilde{A} = A$.

TC5. Based on TC1, we change $\varphi_{\texttt{Eve}}$, the query used in all the available probes, to a compound query:

$$\varphi_{\texttt{Eve}} = \mathsf{canExec}(\texttt{Cluster}, \texttt{Eve}, \texttt{Job}) \wedge \\ \quad \neg\mathsf{isBanned}(\texttt{Cluster}, \texttt{Eve}).$$

This change does not affect the detectability of π_0.

TC6. Based on TC5, we prune **Avail** down to three probes that together are sufficient for proving detectability:

$\langle\{(9),(10),(11)\},\varphi_{\texttt{Eve}}\rangle$
$\langle\{(9),(10)\},\varphi_{\texttt{Eve}}\rangle$
$\langle\{(9),(10),(12)\},\varphi_{\texttt{Eve}}\rangle$

Only the first of these three probes is positive. But unlike the case in TC4, we cannot make use of the minimally positive probe optimization, as the modified $\varphi_{\texttt{Eve}}$ from TC5 is non-monotonic.

7.2 Observations

The query $\varphi_{\texttt{Eve}}$ used in **Avail** in TC1–TC4 does not involve negation, hence there are no conflicting probes to remove. Not surprisingly, enabling the conflicting probe optimization does not decrease the verification time, but it is not a significant overhead either. The query does not involve disjunction either, hence there is only one initial state in TC1–TC4.

In TC1–TC3, **Avail** contains a downward closed probe set containing many redundant probes. The redundant probe optimization effectively decreases both the verification time and the number of calls made to the Datalog engine; in the case of TC3, by a factor of 280.

TC2 is very similar to TC1, but runs 40% faster, because TC1 has a detectable input probe (and hence traverses the entire search space), whereas TC2 has an opaque input probe (and hence stops when the first witness is found).

As TC3 shows, increasing the number of available probes (here by a factor of 8) dramatically decreases performance when no optimizations are used (by a factor of 1130). With optimizations enabled, the verification time increases only by a factor of 8.

TC4 and TC6 show that manually picking the relevant probes is the most effective strategy for decreasing verification times. For example, in the case of TC4, it decreases by a factor of 3150 compared to TC3 in the non-optimized configuration, and a factor 11 compared to the optimized one. In the case of TC6, the speedup factors are even more extreme: 19,000 (vs. TC5, non-optimized) and 150 (vs. TC5, optimized).

In TC5, the negation in $\varphi_{\texttt{Eve}}$ leads to an explosion of the size of **Init** (up to 16,384 from just 1). Here, the conflicting probe optimization is highly effective, as all but one of the initial states are conflicting. Removing them decreases the verification time by a factor of 126 (from over 15 min down to 7.2 s). Of course, this is still much slower than with the manually pruned probe set of TC6, which only takes 50 ms.

To measure the dependency between the size of **Avail** and the verification time, we additionally created two data series: one in which independent trivially positive probes $\langle\{p_i\},p_i\rangle$ are successively added to **Avail**, and a second one in which the added probes are of the form $\langle\{p_i\},z\rangle$ and thus trivially negative. The verification time increases exponentially for the first series, with every additional positive probe $\langle\{p_i\},p_i\rangle$ doubling the time, which is also predicted by Theorem 5.17. In the second series, the time increases linearly with the number of additional irrelevant probes, with each negative probe roughly adding 1.3 ms. This is also to be expected, as negative probes do not cause any branching.

8 Discussion

To recapitulate, we first presented a general framework of probing attacks, defining abstract notions of policy, probe, and adversary characterized by available probes, and based on these, notions of observational equivalence, opacity and detectability. We instantiated this framework to Datalog, a language on which many existing policy languages are based.

It has been an open question whether the problem of opacity in Datalog policies is decidable [3]. We answered this question in the positive by presenting a complete decision procedure for opacity. It works by attempting to construct opacity witnesses, i.e., policies that masquerade as the original policy, but falsify the input probe. We also devised a number of optimization strategies for pruning the search space. Our experimental results on a realistic delegation policy show that these methods are highly effective.

Opacity, as defined here, is a *possibilistic* information flow property. The mere possibility of the existence of an opacity witness suffices to deem an input probe as opaque, no matter how unlikely these witnesses may be. But our algorithm for deciding opacity provides richer information, as it does not merely prove existence of a witness, but actually enumerates all minimal witnesses. The set of minimal witnesses is a finite representation of the infinite set of all witnesses. Hence, by inspecting the minimal witnesses, a user can informally judge the likelihood of opacity or detectability. In theory, the algorithm could also be used to provide results based on a *probablistic* notion of opacity, if witnesses can be mapped to numerical likelihoods (which we believe is generally not feasible in practice, though).

Future work. We see two major limitations in the presented approach. Firstly, although we allow the service policy A_0 to contain arbitrary non-ground clauses, we restrict the set of available probes **Avail** to ground probes. This restriction is reasonable insofar as third-party credentials possessed by adversaries are usually ground, and services may only allow incoming credentials that are ground. However, if the latter condition does not hold, then the adversary could plausibly self-issue non-ground credentials and use them in probes that would make for strictly more powerful probing attacks. Consider, for instance, the two probes

$$\langle \{p \leftarrow \mathsf{isSecretAgent}(x)\}, z\rangle, \text{ and}$$
$$\langle \emptyset, z\rangle.$$

If the first one is positive in A_0 and the second one negative, then the adversary can deduce that there *exists some* secret agent, according to A_0. We leave it to future work to see if the current approach can be extended to deal with non-ground probe sets.

Secondly, the adversary is currently restricted to use a finite set of probes **Avail** for probing attacks. In practice, it would be useful to be able to prove opacity in the context of an adversary who has infinitely many probes to her disposal. Instead of enumerating all the probes, one could finitely specify the infinite probe set via constraints. Perhaps the current algorithm could be applied to an abstract interpretation [11] of the probe set, possibly at the expense of completeness. Again, we defer further investigation to future work.

Finally, in this paper we only provided an upper bound on the complexity of the opacity problem. A deeper analysis would be required to establish a lower bound.

References

[1] A. Anderson. Web Services Profile of XACML (WS-XACML) Version 1.0. *OASIS TC Working Draft*, 2006.

[2] A. Anderson and H. Lockhart. SAML 2.0 Profile of XACML v2. 0. *OASIS Standard*, 2005.

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

[4] 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.

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

[6] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

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

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

[9] 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.

[10] 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.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[12] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, London, UK, 2001. Springer-Verlag.

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

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

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

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

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

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

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

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

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

[22] 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.

[23] 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.

[24] L. Mazar. Using unification for opacity properties. In *In Proceedings of the Workshop on Issues in the Theory of Security (WITS04*, pages 165–176, 2004.

[25] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19:629–679, 1994.

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

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

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

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

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

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