Opacity Analysis in Trust Management Systems

Moritz Y. Becker
Microsoft Research, Cambridge, UK

Masoud Koleini
University of Birmingham, UK

Abstract Trust management systems are vulnerable to so-called probing attacks, which enable an adversary to gain knowledge about confidential facts in the system. We present the first method for deciding if an adversary can gain knowledge about confidential information in a Datalog-based policy.

1 Introduction

In the trust management paradigm [7], authorization rules are specified in a high-level *policy language* (e.g. [18,4,11,17]). Access is granted only if the user's request complies with the policy in conjunction with the user-submitted *credentials*. There is a class of attacks on such systems, called *probing attacks*, that enables an adversary to gain knowledge about confidential information in a service's policy, by submitting a series of *probes*, i.e., access requests together with conditional credentials.

Here is an example of a simple probing attack on a policy written in Binder [11]. The service `Hospital` publishes the policy rule "`Hospital` **says** canCreateAcc(x) **if** `AgeCert` **says** over21(x)". This rule stipulates that any principal x can create a patient account if `AgeCert` says that x is an adult. `Hospital`'s policy also contains confidential facts that are not visible to the adversary `Eve`, for instance, whether `Bob` is a patient or not. (Our use of the term "policy" includes not only rules, but also all authorization-relevant facts.) Suppose `Eve` collaborates with `AgeCert`, and hence can get hold of the `AgeCert`-issued credential "`AgeCert` **says** over21(`Eve`) **if** `Hospital` **says** patient(`Bob`)". She starts her probing attack by submitting this credential together with a request to create a patient account. The service evaluates the corresponding query "`Hospital` **says** canCreateAcc(`Eve`)?" against its policy *in union* with the credential. Suppose it responds by granting access. Next, `Eve` submits a second probe, with the same access request, but with no supporting credentials. This time, access is denied.

From these two probes, `Eve` deduces that the submitted credential must have been crucial in making the access query succeed. But this is only possible if credential's condition "`Hospital` **says** patient(`Bob`)" is true in `Hospital`'s policy. She has therefore *detected* a confidential fact through probing.

Probing attacks present a serious problem to trust management systems, as policies commonly contain confidential information, and it takes little effort to conduct a probing attack with legitimately collected and self-issued credentials. 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 information in credential systems.

This paper presents several significant contributions to the problem space. Firstly, on a foundational level, it provides the first formal framework for probing attacks that makes no assumptions on the structure of policies and of the credential evaluation mechanism, and that is thus general enough to encompass widely different languages such as XACML [18] and DKAL2 [14]. Based on an abstract notion of observational equivalence, this framework specifies precisely what it means for a piece of information to be detectable or opaque (Section 3).

Secondly, we present the first algorithm for checking opacity in policies written in Datalog, which is the basis of many existing policy languages (Section 5). 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 mere existence of a complete decision procedure for opacity in this context is far from obvious.

A particularly 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 construct 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 a degree of likelihood of the property being opaque.

Thirdly, we identify several optimization methods for cutting the high computational cost by pruning the search space, and show empirically that these render the opacity verification problem feasible in medium-sized cases, whereas the straightforward implementation of the algorithm is unusable for other than very small test cases (Section 6). Full proofs and extended examples are included in a technical report [5].

2 Related Work

Probing attacks on credential systems were first mentioned in [13]. 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 [14] is susceptible to probing attacks [3].

Probing attacks were first defined in terms of opacity in [3]. However, in contrast to our general framework (Section 3), their definitions only apply to simple logic-based policy languages, but not to more complex languages such as XACML [18] or Ponder [10], 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. That paper also presents an inference system for analyzing detectability in Datalog policies, but it is incomplete and non-constructive in the sense that it does not map to a terminating algorithm, and thus cannot be used to check opacity. We present the first decision procedure for proving opacity (Section 5).

Research on information flow has mainly focused on stateful, temporal computations (see [19] for an overview). The current setting is very different as there is no notion of state, 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 [15,12], mainly for relational databases. Bonatti et al. have studied the database inference problem in deductive databases [8], 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).

There has been some work on formalizing and enforcing safety in Automated Trust Negotiaton (ATN) protocols [22,21], i.e., the property that no information about the presence or absence of credentials is prematurely leaked during a credential exchange [20]. This problem is quite different from the one we are considering here; e.g., we are interested in the confidentiality of internal properties of the policy rather than that of submitted credentials, and our credentials are not mere attributes, but may be conditional and may affect policy evaluation results.

3 A Framework for Probing Attacks

3.1 Abstract Framework

This section establishes the fundamental concepts for reasoning about probing attacks in credential systems.

Definition 1 (Policy language, probe). *A* policy language *is a triple* $(\mathbf{Pol}, \mathbf{Prb}, \vdash)$*, where* $\mathbf{Pol}$ *and* $\mathbf{Prb}$ *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*.*

Although Definition 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 1, we briefly sketch how it would be instantiated to the concrete policy languages SecPAL [4], DKAL2 [14], and XACML [18].

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 "$\langle$*Principal*$\rangle$ **says** $\langle$*Fact*$\rangle$". 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 [17], Cassandra [6], SD3 [16], and Binder [11].

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 $\longleftarrow$ B canWrite" submitted by Eve would be imported as the infon term "B canWrite $\rightarrow$ 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 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 be transformed before evaluation. For example, the client-supplied Attribute may be written in SAML and would first 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 typically determined, firstly, by which credentials the adversary possesses or can create, and, secondly, by which queries the service allows her to run. For instance, SecPAL services define an Authorization Query Table 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 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 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.

Definition 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. in Definition 3. Hence an active adversary cannot distinguish policies that are observationally equivalent.

We are interested in whether the adversary can infer 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 4 (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 $\Phi \subseteq \mathbf{Pol}$ *is* opaque in $A \in \mathbf{Pol}$ *iff it is not detectable in A, or equivalently, iff* $\exists A' \in \mathbf{Pol}: \; A \equiv A' \; \wedge \; \neg\Phi(A')$*.*

The definitions established in this section so far provide a general framework for reasoning about probing attacks in credential systems. For specific 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 in Datalog, which will also help illustrate the definitions above.

3.2 Datalog-Based Policies

Datalog 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 it (e.g. [4,17,6,16,11]). Reasoning techniques and analysis tools for Datalog therefore apply to a wide range of policy languages. We only give a very brief overview of Datalog. (For a more careful introduction, see e.g. [9].)

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 P_i are *atoms* of the form $p(\vec{e})$ (where p is a predicate symbol, and $\vec{e}$ a sequence of variables and constants). (We usually omit the arrow if $n = 0$.) We write $\mathbf{hd}(a)$ to denote a's *head* P_0 and $\mathbf{bd}(a)$ to denote its *body* $\vec{P} = \langle P_1, ..., P_n \rangle$. 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 **true**, **false** or a ground (i.e., variable-free) boolean formula (i.e., involving connectives $\neg$, $\wedge$ and $\vee$) over atoms P. We write **Qry** to denote the set of all queries. A query φ is evaluated with respect to a set A of assertions. For atomic $\varphi = P$, we define that $A \vdash P$ holds iff there exists a ground (i.e., variable-free) instance $P \leftarrow \vec{P}$ of some clause in A and $A \vdash P_i$ for all $P_i \in \vec{P}$. The non-atomic cases are defined in the standard way, e.g. $A \vdash \neg\varphi$ iff $A \nvdash \varphi$.

Now we can instantiate the abstract Definitions 1 and 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 5 (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 by* $A_0 \vdash \langle A, \varphi \rangle \iff A_0 \cup A \vdash \varphi$*.*

Definition 6 (Adversary, Datalog alikeness). *An* adversary *is defined by a set* **Avail** $\subseteq$ **Prb** *and a unary predicate* **Visible** $\subseteq$ **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* $\mathbf{Visible}(A_1) = \mathbf{Visible}(A_2)$*.*

Definitions 5 and 6 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 (usually one that is not in **Avail**) is positive or negative.

Definition 7 (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 4, with the predicate Φ instantiated to $\{A \subseteq \mathbf{Cls} \mid A \vdash \pi\}$.

4 Example

We illustrate the definitions from the previous section using an example of an authorization policy written in Datalog. The example also serves as the basis for the test cases in Section 6. 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. The policies consist of the following seven clauses:

$$\mathsf{canExe}(\texttt{Clstr}, x, j) \leftarrow \mathsf{mem}(\texttt{Clstr}, x), \mathsf{owns}(\texttt{Clstr}, x, j), \mathsf{canRd}(\texttt{Data}, \texttt{Clstr}, j). \tag{1}$$
$$\mathsf{owns}(\texttt{Clstr}, x, j) \leftarrow \mathsf{owns}(y, x, j), \mathsf{isTTP}(\texttt{Clstr}, y). \tag{2}$$
$$\mathsf{mem}(\texttt{Clstr}, x, j) \leftarrow \mathsf{mem}(y, x, j), \mathsf{isTTP}(\texttt{Clstr}, y). \tag{3}$$
$$\mathsf{canRd}(\texttt{Data}, x, j) \leftarrow \mathsf{canRd}(y, x, j), \mathsf{owns}(\texttt{Data}, y, j). \tag{4}$$
$$\mathsf{owns}(\texttt{Data}, x, j) \leftarrow \mathsf{owns}(y, x, j), \mathsf{isTTP}(\texttt{Data}, y). \tag{5}$$
$$\mathsf{isTTP}(\texttt{Clstr}, \texttt{CA}). \tag{6}$$
$$\mathsf{isTTP}(\texttt{Data}, \texttt{CA}). \tag{7}$$

Here, 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. For instance, $\mathsf{canExe}(\texttt{Clstr}, x, j)$ intuitively means that `Clstr` says that x can execute job j.

According to Clause (1), anyone who is a member and owns a job (according to `Clstr`) can execute that job (according to `Clstr`), if the data center `Data` allows `Clstr` to read the data associated with that job. `Clstr` delegates authority over job ownership and membership to trusted third parties (2)–(3). The next clause implements a variant of discretionary access control: data center `Data` stipulates that owners y of data associated with job j can delegate read access to this data to other principals x (4). Just like `Clstr`, `Data` delegates authority over ownership to third parties it trusts (5). Finally, both `Clstr` and `Data` specify certificate authority `CA` as a trusted third party (6)–(7).

`Clstr` has an interface that allows users to submit a job execution request. When some user `Eve` requests to execute a job `Job`, the corresponding query

$$\varphi_{\texttt{Eve}} = \mathsf{canExe}(\texttt{Clstr}, \mathsf{Eve}, \mathsf{Job}) \tag{8}$$

is evaluated against the policy consisting of the clauses (1)–(7), in union with the (possibly empty) set of credentials submitted by `Eve` together with the request.

`Eve`, who plays the role of the adversary in our scenario, possesses four credentials:

$$\mathsf{owns}(\texttt{CA}, \mathsf{Eve}, \mathsf{Job}). \tag{9}$$
$$\mathsf{mem}(\texttt{CA}, \mathsf{Eve}). \tag{10}$$
$$\mathsf{canRd}(\mathsf{Eve}, \texttt{Clstr}, \mathsf{Job}). \tag{11}$$
$$\mathsf{canRd}(\mathsf{Eve}, \texttt{Clstr}, \mathsf{Job}) \leftarrow \mathsf{mem}(\texttt{Clstr}, \mathsf{Bob}). \tag{12}$$

Credentials (9)–(10) are issued by `CA`, and the other two are self-issued. `Eve` is interested in finding out if `Bob` is a member, according to `Clstr`'s policy. Of course, she does not have the authority to query this fact directly, so instead she hopes to be able to detect this fact using (12) in particular, stating that she is willing to give `Clstr` read access, provided that `Bob` is a member of `Clstr`.

Let A_0 be the policy consisting of clauses (1)–(7), and $A_{\texttt{Eve}}$ be the set of clauses (9)–(12). $A_{\texttt{Eve}}$ 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: **Avail** $= \{\langle A, \varphi_{\texttt{Eve}}\rangle \mid A \subseteq A_{\texttt{Eve}}\}$. For simplicity, we assume that **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.

We have $A_0 \vdash \langle A_{\texttt{Eve}}, \varphi_{\texttt{Eve}}\rangle$, 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 `Clstr`. Hence `Data` allows `Eve` to delegate read access to `Clstr` using (11). Furthermore, (10) is sufficient for proving `Eve`'s membership to `Clstr`, hence all body atoms of (1) are satisfied, which implies $\varphi_{\texttt{Eve}}$.

We also have $A_0 \vdash \langle\{(9)\text{–}(11)\}, \varphi_{\texttt{Eve}}\rangle$, since the derivation above only makes use of the clauses (9)–(11). But $A_0 \nvdash \langle A, \varphi_{\texttt{Eve}}\rangle$, 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.

All policies A_0' that are observationally equivalent to A_0, i.e., that exhibit the same behaviour as observed above, satisfy the property that $A_0 \nvdash \mathsf{mem}(\texttt{Clstr}, \texttt{Bob})$. For suppose the contrary were the case. We observed 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 the observation that $\varphi_{\texttt{Eve}}$ does *not* hold in $\{(9),(10),(12)\}$.

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

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.

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

Overview. The algorithm is succinctly specified as the transition system in Fig. 1, but it is actually rather involved. We first give a high-level roadmap of the algorithm before proceeding to the details.

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, the algorithm attempts to construct such an *opacity witness* A_0'. Conversely, to prove detectability, it proves that no such A_0' exists.

A *state* in the transition system is a triple of the form $\langle \Pi^+, \Pi^-, A_1\rangle$, and the (INIT) rule in Fig. 1 defines the set **Init** of *initial states.* Intuitively, an initial state is populated with sets Π^+, Π^- of probes that are required to be positive (negative,

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

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

Figure 1. Transition system for verifying opacity.

respectively) in the opacity witness A_0' to be constructed. At each (PROBE) transition, the system considers and discards one positive probe in Π^+, and adds a set of clauses to the *witness candidate* $A_1 \subseteq \mathbf{Cls}$. Theorem 1 states that π_0 is opaque iff the transition system reaches a state of the form $\langle \emptyset, \Pi^-, A_0'\rangle$, starting from some initial state. Furthermore, A_0' will be an opacity witness.

Our results also show that opacity checking is decidable. This is nontrivial, as the definition of opacity is quantified over the infinite set of all policies; and many other simple-looking quantified properties such as containment are undecidable. (Note that the set of predicate symbols and constants may be infinite.) The decidability of opacity checking essentially stems from a sort of topological compactness property of the set of policies A_0' that are observationally equivalent to A_0. More precisely, even though there may be infinitely many candidates for A_0', we only ever need to consider a finite number of them.

5.1 Initial States

The initial states **Init**, defined declaratively by (INIT) in Fig. 1, are produced by transforming all available probes into equivalent disjunction- and negation-free ones. Consider a disjunctive probe $\pi = \langle A, \varphi_1 \vee \varphi_2\rangle \in \mathbf{Avail}$ that is positive in A_0. The algorithm 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 positive 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 function $\mathbf{flatten}_{A_0}$ (defined below) applied to **Avail** first performs the mentioned conversion of negative probes into equivalent positive ones. It then splits each probe into disjuncts of atomic and negated atomic queries. Finally, it produces a cartesian product of all these disjuncts that keeps the atomic and negated queries apart. The result is a set of pairs of disjunction-free probe sets; each such pair (Π^+, Π^-) corresponds to a disjunctive search branch. The problem of finding a policy A_0' that is observationally equivalent to A_0 (*cf.* Def. 3) can then be reduced to finding an A_0' and picking a pair $(\Pi^+, \Pi^-) \in \mathbf{flatten}_{A_0}(\mathbf{Avail})$ such that all probes in Π^+ are positive, and all probes in Π^- are negative in A_0'.

In the following, we write $\mathbf{dnf}(\varphi)$ to denote the disjunctive normal form of a query φ, represented as a set of pairs (S^+, S^-) of sets of atoms. For instance, if $\varphi = (p \wedge q \wedge \neg s) \vee (\neg p \wedge \neg q \wedge s)$, then $\mathbf{dnf}(\varphi) = \{(\{p, q\}, \{s\}), (\{s\}, \{p, q\})\}$.

Definition 8 (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:*

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

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

Apart from the observational equivalence $A_0' \equiv A_0$, opacity of π_0 in A_0 additionally requires that π_0 be negative in A_0'. Let $\pi_0 = \langle A, \varphi\rangle$. This is equivalent to finding a pair $(S^+, S^-) \in \mathbf{dnf}(\neg\varphi)$ such that $A_0' \vdash \langle A, \bigwedge S^+\rangle$ and $A_0' \nvdash \langle A, \bigvee S^-\rangle$.

We can then reduce the problem of proving opacity of π_0 in A_0 to constructing an A_0' for some $(\Pi_0^+, \Pi_0^-) \in \mathbf{flatten}_{A_0}(\mathbf{Avail})$ and $(S^+, S^-) \in \mathbf{dnf}(\neg\varphi)$ such that all probes in $\Pi^+ = \Pi_0^+ \cup \{\langle A, \bigwedge S^+\rangle\}$ are positive, and all probes in $\Pi^- = \Pi_0^- \cup \{\langle A, \bigvee S^-\rangle\}$ are negative in A_0'. If such an A_0' exists, we say that A_0' is a *witness* (for the opacity of π_0 in A_0). We call Π^+ a set of *positive probe requirements*, and Π^- a set of *negative probe requirements*.

Furthermore, from the alikeness condition $A_0' \simeq A_0$, any witness must contain $\mathbf{Visible}(A_0)$. Hence (INIT) picks $\mathbf{Visible}(A_0)$ as the initial witness candidate for each initial state. The last premise in (INIT) filters out those witnesses candidates that fail to make all probes in Π^- negative.

These observations are formalized in Lemma 1, stating the correctness of (INIT).

Lemma 1. *π_0 is opaque in A_0 iff there exist $\langle \Pi^+, \Pi^-, \mathbf{Visible}(A_0)\rangle \in \mathbf{Init}$ and $A_0' \subseteq \mathbf{Cls}$ such that $A_0' \supseteq \mathbf{Visible}(A_0)$ and $\forall \pi \in \Pi^+ : A_0' \vdash \pi$ and $\forall \pi \in \Pi^- : A_0' \nvdash \pi$.*

5.2 Finding Minimal Witnesses

Given **Init**, we now have to find a witness A_0' that satisfies the requirements from Lemma 1. Consider an initial or intermediate state $\langle \Pi^+ \cup \{\pi\},\ \Pi^-,\ A'\rangle$. The transition rule (PROBE) from Fig. 1 picks the positive probe requirement π and adds to the current witness candidate A' a set A'' of clauses such that $A' \cup A'' \vdash \pi$. The monotonicity of $\vdash$ guarantees that adding A'' does not make any previously considered positive probe requirement negative. However, adding clauses may make negative probe requirements in Π^- positive, so we need to check $\forall \pi' \in \Pi^- : A' \cup A'' \nvdash \pi'$. If this fails, the algorithm backtracks to try out a different A''. Otherwise, the transition succeeds and produces the new state $\langle \Pi^+,\ \Pi^-,\ A' \cup A''\rangle$. If we reach a finite state, i.e. one where Π^+ is empty, then π_0 is opaque, by Lemma 1, and moreover, the witness candidate of the final state is a genuine witness. This informally shows that the algorithm is sound.

Minimality. To ensure completeness, we have to consider *all* candidate extensions A'' such that $A' \cup A'' \vdash \pi$. It turns out that we can ignore A' and simply consider all A'' such that $A'' \vdash \pi$ (which then implies $A \cup A'' \vdash \pi$). However, there may be infinitely many such A''. At the same time, 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 the goal π_0 is *not* opaque (i.e., detectable). We certainly do not want infinite branching. Fortunately, it turns out that we do not need to compute all candidate extensions. Instead, we only compute the *minimal* ones. This notion of minimality is based on Datalog containment.

Definition 9 (Containment). *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: $A \vdash \langle S, P\rangle \Rightarrow A' \vdash \langle S, P\rangle$.*

So, to be more precise, the candidate extensions actually considered by the algorithm form a *finite* set S such that

– $\forall A'' \in S: \ A'' \vdash \pi$, and
– $\forall \tilde{A}'' \subseteq \mathbf{Cls}: \quad \tilde{A}'' \vdash \pi \Rightarrow \exists A'' \in S : A'' \preceq \tilde{A}''$.

This property has two significant ramifications. Firstly, it ensures termination, since S is finite for each considered positive probe requirement π; furthermore, each initial state only has finitely many positive probe requirements, and there are only finitely many initial states in **Init**.

Secondly, it ensures that the algorithm is complete: consider any $\tilde{A}'_0$ that makes all positive probe requirements Π^+ of some initial state positive. Then there exists A'_0 constructed by iteratively adding a minimal extension for each $\pi \in \Pi^+$, such that A'_0 also makes all probes in Π^+ positive and $A'_0 \preceq \tilde{A}'_0$. Therefore, if $\tilde{A}'_0$ is a genuine witness (i.e., it also makes all negative probe requirements in Π^- negative) then A'_0 is also a genuine witness, by anti-monotonicity of $\nvdash$. Hence if there is a genuine witness, the algorithm will find one that is at least as small, in finite time.

It remains to explain how (PROBE) computes the minimal extensions A''.

Relevant subprobes. To gain an intuition for this process, it helps to ask the question "how could $\pi = \langle A, \varphi\rangle$ possibly be positive in some policy 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 in union with A''. 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, i.e., $A'' \vdash \langle \tilde{A}, \varphi\rangle$ but $A'' \nvdash \langle \tilde{A}', \varphi\rangle$ for all $\tilde{A}' \subsetneq \tilde{A}$.

We need to consider all of these $2^{|A|}$ possible cases, since, as we shall see, each different choice of $\tilde{A}$ results in a different set of minimal witness extensions A''. This source of branching is reflected in the condition $\tilde{A} \subseteq A$ in the transition rule (PROBE) in Fig. 1.

Derivation order. Having chosen $\tilde{A} \subseteq A$ to be relevant, there may still be multiple minimal solutions for A'' that makes $\pi = \langle A, \bigwedge \vec{P}\rangle$ positive. 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, and not even a minimal one. In general, A'' may contain the body atoms of just 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 1. Suppose $\varphi = \vec{P} = z$ and $\tilde{A} = \{p \leftarrow q.,\ r \leftarrow s.,\ u \leftarrow v.\}$. 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. Again, here are two of the 13 solutions:

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

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.

□

It turns out that this observation holds in the general case. Given a particular $\tilde{A} \subseteq A$, we can prove that 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.

Theorem 1 (Soundness and completeness). *π_0 is opaque in A_0 iff there exist $\sigma_0 \in$ **Init**, $\Pi^- \subseteq$ **Prb** and $A'_0 \subseteq$ **Cls** such that $\sigma_0 \to^* \langle \emptyset, \Pi^-, A'_0 \rangle$.*

Theorem 2. *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 $n = |A|$.*

Theorem 2 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 an implementation of Datalog's evaluation relation $\vdash$. It is not highly optimized, and 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, which helps to overcome some of the limitations of the strict possibilistic (as opposed to probabilistic) concept of opacity.

For example, we added the atomic clause $\mathsf{mem}(\mathtt{Clstr}, \mathtt{Bob})$ to the policy in Section 4, and expected the fact that Bob now *is* a member to be detectable by `Eve`. 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 $\mathsf{mem}(\mathtt{Clstr}, \mathtt{Bob})$ is derivable.

However, the prototype (correctly) reports that $\langle \emptyset, \mathsf{mem}(\mathtt{Clstr}, \mathtt{Bob}) \rangle$ is opaque. A closer look at the produced witnesses reveals that they all contain the rather "improbable" clause "$\mathsf{mem}(\mathtt{Clstr}, \mathtt{Bob}) \leftarrow \mathsf{mem}(\mathtt{CA}, \mathtt{Eve}), \mathsf{owns}(\mathtt{CA}, \mathtt{Eve}, \mathtt{Job})$".

Indeed, we can prove this hypothesis with our prototype: the weakened input probe $\langle \{(9), (10)\}, \mathsf{mem}(\mathtt{Clstr}, \mathtt{Bob}) \rangle$ *is* detectable. Thus, the constructiveness of the algorithm enabled us to form the informal judgement that the original input probe was detectable *with a high likelihood.*

Example 2. Here is another example that shows how the tool can be used interactively. Let $\pi_1 = \langle \{b \leftarrow a.,\ d \leftarrow c.\}, e \rangle$, $\pi_2 = \langle \emptyset, \neg a \rangle$, and $\pi_1 = \langle \{d\}, \neg e \rangle$. Suppose that all three probes are positive in A_0, and **Visible**(A_0) $= \emptyset$. What does this tell us about c and a in A_0? This example is interesting because the answer is not obvious on casual inspection and may be somewhat surprising.

We start with the obvious goal probes $\langle \emptyset, c \rangle$ and $\langle \emptyset, a \rangle$. The tool reports that $\langle \emptyset, c \rangle$ is detectable (i.e., c must be true in A_0), but that $\langle \emptyset, a \rangle$ is opaque. For the latter analysis, it also reports that only one minimal witness exists, which includes the clause $a \leftarrow d$. And indeed, the goal probe $\langle \{d\}, a \rangle$ is found to be detectable, hence we can infer that $A_0 \cup \{d\} \vdash a$. (The example is small enough that the interested reader may manually retrace the steps of the algorithm to verify these results.) $\square$

As Theorem 2 indicates, traversing the entire transition system would be infeasible even for small examples. We devised and implemented a number of optimization methods for effectively pruning the search tree. For lack of space, we describe them only very briefly. Full descriptions and proofs of correctness are found in [5].

Order independence. 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. 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.

Redundant probes. The sets Π^+ and Π^- in an initial state often contain many pairs of probes $\pi_1 = \langle A_1, \varphi_1 \rangle$, $\pi_2 = \langle A_2, \varphi_2 \rangle$ such that $\pi_1 \subseteq \pi_2$ (i.e., $\varphi_1 = \varphi_2 \wedge A_1 \subseteq A_2$). 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^-. In general, 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.

Conflicting probes. 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.

Experimental results. To gain a better understanding of the scalability of the opacity checking algorithm, with and without optimization methods, we performed a number of performance tests. We only briefly summarize our findings and refer to the technical report [5] for reproducible details on all test cases and performance plots.

The performance tests are based on the policy from Section 4. The policy is arguably small; however, this fact does not weaken our results, as it is easy to see that the computation time is essentially independent of the size of the policy A_0. The significant parameter with respect to computation time is **Avail**.

We found that the computation time doubles with each irrelevant, trivially positive probe $\langle \{p_i\}, p_i \rangle$ being added to **Avail**, which is predicted by Theorem 2. Successively adding irrelevant, trivially negative probes $\langle \{p_i\}, z \rangle$ only caused a linear increase (+1.3 ms per additional probe). This is also to be expected, as negative probes do not cause any branching.

We then explored several variations of the basic scenario from Section 4, running each test case with different combinations of the optimization methods enabled. (The order independence method was enabled in all runs.)

We found that enabling the optimization methods improved performance in all cases, apart from those where **Avail** was manually reduced to only the relevant probes. Even in the latter cases, enabling optimization did not add any significant overhead. The automated optimization methods led to dramatic improvements that were particularly noticeable in the more complex test cases, with speedup factors between 126 and 280. Furthermore, they also significantly improved scalability; for example, increasing the size of **Avail** from 16 to 128 increased the computation time by a factor of 1130 in the unoptimized case, but only by a factor 8 with the optimizations enabled.

Not surprisingly, manually picking only the relevant probes was the most effective strategy for improving performance, with speedup factors between 150 and 19,000. This suggests that significant performance gains can be expected from more sophisticated pruning methods.

The size of **Avail** varied between 16 and 128. All test cases completed in less than one second (with all optimizations enabled, on a standard workstation), apart from the most complex one, which took 7.2 s (vs. 15 min unoptimized) and 150,000 (vs. 7 million) Datalog query evaluations. In the latter test case, the probe queries contained negation, which led to more than 16,000 initial states.

Our results suggest that checking opacity using our tool is feasible in many practical cases, given that the analysis is almost independent of the size of the policy (which may well have millions of clauses), and that it seems reasonable to restrict analysis to adversaries that have no more than about a hundred different probes to their disposal.

7 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 show that these methods are highly effective.

Opacity is a *possibilistic* information flow property. The mere possibility of the existence of an opacity witness suffices to deem an input probe opaque, no matter how unlikely these witnesses may be. But our algorithm for deciding opacity provides richer information, as it does not merely prove the 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. Our prototype includes an interface that lets the security analyst browse and inspect the witnesses, thereby enabling her to informally judge the likelihood of opacity or detectability. We believe that this is a more useful approach in practice than ascribing numerical probabilities to the witnesses.

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*, 2007.
5. M. Y. Becker and M. Koleini. Information leakage in datalog-based trust management systems. Technical Report MSR-TR-2011-11, Microsoft Research, 2011.
6. M. Y. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *IEEE Computer Security Foundations*, pages 139–154, 2004.
7. M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *IEEE Symposium on Security and Privacy*, pages 164–173, 1996.
8. P. Bonatti, S. Kraus, and V. Subrahmanian. Foundations of secure deductive databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):406–422, 1995.
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. 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.
11. J. Detreville. Binder, a logic-based security language. In *IEEE Symposium on Security and Privacy*, pages 105–113, 2002.
12. C. Farkas and S. Jajodia. The inference problem: a survey. *ACM SIGKDD Explorations Newsletter*, 4(2):6–11, 2002.
13. Y. Gurevich and I. Neeman. DKAL: Distributed-knowledge authorization language. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 149–162, 2008.
14. Y. Gurevich and I. Neeman. DKAL 2 – a simplified and improved authorization language. Technical Report MSR-TR-2009-11, Microsoft Research, 2009.
15. S. Jajodia and C. Meadows. Inference problems in multilevel secure database management systems. *Information Security: An Integrated Collection of Essays*, 1995.
16. 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.
17. 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.
18. OASIS. *eXtensible Access Control Markup Language (XACML) Version 2.0 core specification*, 2005.
19. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.
20. W. Winsborough and N. Li. Safety in automated trust negotiation. *ACM Transactions on Information and System Security (TISSEC)*, 9(3), 2006.
21. W. H. Winsborough and N. Li. Towards practical automated trust negotiation. In *IEEE International Workshop on Policies for Distributed Systems and Networks*, 2002.
22. W. H. Winsborough, K. E. Seamons, and V. E. Jones. Automated trust negotiation. In *DARPA Information Survivability Conference and Exposition*, volume 1, 2000.