Securing Databases from Probabilistic Inference

Marco Guarnieri
Institute of Information Security
Department of Computer Science
ETH Zurich, Switzerland
marco.guarnieri@inf.ethz.ch

Srdjan Marinovic
The Wireless Registry, Inc.
Washington DC, US
srdjan@wirelessregistry.com

David Basin
Institute of Information Security
Department of Computer Science
ETH Zurich, Switzerland
basin@inf.ethz.ch

***Abstract*—Databases can leak confidential information when users combine query results with probabilistic data dependencies and prior knowledge. Current research offers mechanisms that either handle a limited class of dependencies or lack tractable enforcement algorithms. We propose a foundation for Database Inference Control based on PROBLOG, a probabilistic logic programming language. We leverage this foundation to develop ANGERONA, a provably secure enforcement mechanism that prevents information leakage in the presence of probabilistic dependencies. We then provide a tractable inference algorithm for a practically relevant fragment of PROBLOG. We empirically evaluate ANGERONA's performance showing that it scales to relevant security-critical problems.**

I. INTRODUCTION

Protecting the confidentiality of sensitive data stored in databases requires protection from both *direct* and *indirect* access. The former happens when a user observes query results, and the latter happens when a user infers sensitive information by combining results with external information, such as data dependencies or prior knowledge. Controlling indirect access to data is often referred to as *Database Inference Control* [30] (DBIC). This topic has attracted considerable attention in recent years, and current research considers different sources of external information, such as the database schema [15], [38], [40], [43], [57], [63], [64], the system's semantics [38], statistical information [3], [16], [22]–[24], exceptions [38], error messages [45], user-defined functions [45], and data dependencies [10], [12], [55], [56], [67], [68], [72].

An important and relevant class of data dependencies are probabilistic dependencies, such as those found in genomics [44], [48], [50], social networks [41], and location tracking [53]. Attackers can exploit these dependencies to infer sensitive information with high confidence. To effectively prevent probabilistic inferences, DBIC mechanisms should (1) support a large class of probabilistic dependencies, and (2) have tractable runtime performance. The former is needed to express different attacker models. The latter is necessary for mechanisms to scale to real-world databases.

Most existing DBIC mechanisms support only precise data dependencies [10], [12], [67], [68], [72] or just limited classes of probabilistic dependencies [14], [15], [40], [46], [55], [56], [71]. As a result, they cannot reason about the complex probabilistic dependencies that exist in many realistic settings. Mardziel et al.'s mechanism [52] instead supports arbitrary probabilistic dependencies, but no complexity bounds have been established and their algorithm appears to be intractable.

Contributions. We develop a tractable and practically useful DBIC mechanism based on probabilistic logic programming.

First, we develop ATKLOG, a language for formalizing users' beliefs and how they evolve while interacting with the system. ATKLOG builds on PROBLOG [20], [21], [31], a state-of-the-art probabilistic extension of DATALOG, and extends its semantics by building on three key ideas from [18], [47], [52]: (1) users' beliefs can be represented as probability distributions, (2) belief revision can be performed by conditioning the probability distribution based on the users' observations, and (3) rejecting queries as insecure may leak information. By combining DATALOG with probabilistic models and belief revision based on users' knowledge, ATKLOG provides a natural and expressive language to model users' beliefs and thereby serves as a foundation for DBIC in the presence of probabilistic inferences.

Second, we identify acyclic PROBLOG programs, a class of programs where probabilistic inference's data complexity is PTIME. We precisely characterize this class and develop a dedicated inference engine. Since PROBLOG's inference is intractable in general, we see acyclic programs as an essential building block to effectively using ATKLOG for DBIC.

Finally, we present ANGERONA[1], a novel DBIC mechanism that secures databases against probabilistic inferences. We prove that ANGERONA is secure with respect to any ATKLOG-attacker. In contrast to existing mechanisms, ANGERONA provides precise tractability and completeness guarantees for a practically relevant class of attackers. We empirically show that ANGERONA scales to relevant problems of interest.

Structure. In §II, we illustrate the security risks associated with probabilistic data dependencies. In §III, we present our system model, which we formalize in §IV. We introduce ATKLOG in §V and in §VI we present our inference engine for acyclic programs. In §VII, we present ANGERONA. We discuss related work in §VIII and draw conclusions in §IX. An extended version of this paper with proofs of all results is available at [36], whereas a prototype of our enforcement mechanism is available at [37].

[1]Angerona is the Roman goddess of silence and secrecy, and She is the keeper of the city's sacred, and secret, name.

II. Motivating Example

Hospitals and medical research centres store large quantities of health-related information for purposes ranging from diagnosis to research. As this information is extremely sensitive, the databases used must be carefully secured [13], [28]. This task is, however, challenging due to the dependencies between health-related data items. For instance, information about someone's hereditary diseases or genome can be inferred from information about her relatives. Even seemingly non-sensitive information, such as someone's job or habits, may leak sensitive health-related information such as her predisposition to diseases. Most of these dependencies can be formalized using probabilistic models developed by medical researchers.

Consider a database storing information about the smoking habits of patients and whether they have been diagnosed with lung cancer. The database contains the tables $patient$, $smokes$, $cancer$, $father$, and $mother$. The first table contains all patients, the second contains all regular smokers, the third contains all diagnosed patients, and the last two associate patients with their parents. Now consider the following probabilistic model: (a) every patient has a 5% chance of developing cancer, (b) for each parent with cancer, the likelihood that a child develops cancer increases by 15%, and (c) if a patient smokes regularly, his probability of developing cancer increases by 25%. We intentionally work with a simple model since, despite its simplicity, it illustrates the challenges of securing data with probabilistic dependencies. We refer the reader to medical research for more realistic probabilistic models [4], [69].

The database is shared between different medical researchers, each conducting a research study on a subset of the patients. All researchers have access to the $patient$, $smokes$, $father$, and $mother$ tables. Each researcher, however, has access only to the subset of the $cancer$ table associated with the patients that opted-in to his research study. We want to protect our database against a malicious researcher whose goal is to infer the health status of patients not participating in the study. This is challenging since restricting direct access to the $cancer$ table is insufficient. Sensitive information may be leaked even by queries involving only authorized data. For instance, the attacker may know that the patient $Carl$, which has not disclosed his health status, smokes regularly. From this, he can infer that $Carl$'s probability of developing lung cancer is, at least, 30%. If, additionally, $Carl$'s parents opted-in to the research study and both have cancer, the attacker can directly infer that the probability of $Carl$ developing lung cancer is 60% by accessing his parents' information.

Security mechanisms that ignore such probabilistic dependencies allow attackers to infer sensitive information. An alternative is to use standard DBIC mechanisms and encode all dependencies as precise, non-probabilistic, dependencies. This, however, would result in an unusable system. Medical researchers, even honest ones, would be able to access the health-related status only of those patients whose relatives also opted-in to the user study, independently of the amount of leaked information, which may be negligible. Hence, to secure the database and retain usability, it is essential to reason about the probabilistic dependencies.

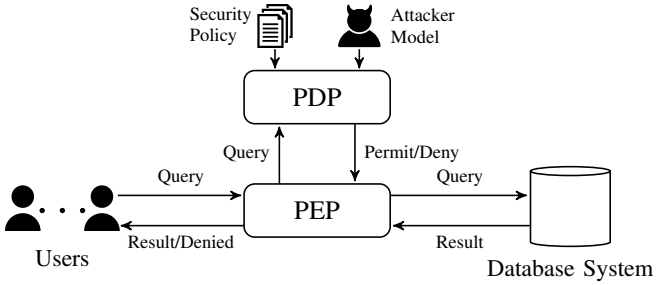


Fig. 1: System model.

III. System Model

Figure 1 depicts our system model. Users interact with two components: a database system and an inference control system, which consists of a Policy Decision Point (PDP) and a Policy Enforcement Point (PEP). We assume that all communication between users and the components and between the components themselves is over secure channels.

Database System. The database system manages the system's data. Its state is a mapping from tables to sets of tuples.

Users. Each user has a unique account used to retrieve information from the database system by issuing `SELECT` commands. Note that these commands do not change the database state. This reflects settings where users have only read-access to a database. Each command is checked by the inference control system and is executed if and only if the command is authorized by the security policy.

Security policy. The system's security policy consists of a set of *negative permissions* specifying information to be kept secret. These permissions express bounds on users' beliefs, formalized as probability distributions, about the actual database content. Negative permissions are formalized using commands of the form `SECRET` q `FOR` u `THRESHOLD` l, where q is a query, u is a user identifier, and l is a rational number, $0 \leq l \leq 1$. This represents the requirement that "A user u's belief in the result of q must be less than l." Namely, the probability assigned by u's belief to q's result must be less than l. Requirements like "A user u is not authorized to know the result of q" can be formalized as `SECRET` q `FOR` u `THRESHOLD 1`. The system also supports commands of the form `SECRET` q `FOR USERS NOT IN` $\{u_1, \ldots, u_n\}$ `THRESHOLD` l, which represents the requirement that "For all users $u \notin \{u_1, \ldots, u_n\}$, u's belief in the result of q must be less than l."

Attacker. An attacker is a system user with an assigned user account, and each user is a potential attacker. An attacker's goal is to violate the security policy, that is, to read or infer information about one of the `SECRET`s with a probability of at least the given threshold.

An attacker can interact with the system and observe its behaviour in response to his commands. Furthermore, he

can reason about this information and infer information by exploiting domain-specific relationships between data items. We assume that attackers know the database schema as well as any integrity constraints on it.

Attacker Model. An attacker model represents each user's initial beliefs about the actual database state and how he updates his beliefs by interacting with the system and observing its behaviour in response to his commands. These beliefs may reflect the attacker's knowledge of domain-specific relationships between the data items or prior knowledge.

Inference Control System. The inference control system protects the confidentiality of database data. It consists of a PEP and a PDP, configured with a security policy P and an attacker model ATK. For each user, the inference control system keeps track of the user's beliefs according to ATK.

The system intercepts all commands issued by the users. When a user u issues a command c, the inference control system decides whether u is authorized to execute c. If c complies with the policy, i.e., the users' beliefs still satisfy P even after executing c, then the system forwards the command to the database, which executes c and returns its result to u. Otherwise, it raises a *security exception* and rejects c.

IV. Formal Model

A. Database Model

We introduce here background and notation for databases and queries. Our formalization follows [2].

Let $\mathcal{R}$ be a countably infinite set representing identifiers of relation schemas. A *database schema* D is a pair $\langle \Sigma, \mathbf{dom} \rangle$, where Σ is a first-order signature and $\mathbf{dom}$ is a fixed domain. For simplicity, we consider just a single domain. Extensions to the many-sorted case are straightforward [2]. The signature Σ consists of a set of *relation schemas* $R \in \mathcal{R}$, each schema with arity $|R|$, and one constant symbol for each constant in $\mathbf{dom}$. We interpret constants by themselves in the semantics.

A *state* s of D is a finite Σ-structure with domain $\mathbf{dom}$ that interprets each relation schema R by an $|R|$-ary relation over $\mathbf{dom}$. We denote by Ω_D the set of all states. Given a schema $R \in D$, $s(R)$ denotes the tuples that belong to (the interpretation of) R in the state s. We assume that the domain $\mathbf{dom}$ is finite, as is standard for many application areas combining databases and probabilistic reasoning [20], [33], [48], [65]. In this case, the set of all states Ω_D is finite.

A *query* q over a schema D is of the form $\{\overline{x} \mid \phi\}$, where $\overline{x}$ is a sequence of variables, ϕ is a relational calculus formula over D, and ϕ's free variables are those in $\overline{x}$. A *boolean query* is a query $\{ \mid \phi\}$, also written as ϕ, where ϕ is a sentence. The result of executing a query q on a state s, denoted by $[q]^s$, is a boolean value in $\{\top, \bot\}$, if q is a boolean query, or a set of tuples otherwise. Furthermore, given a sentence ϕ, $[\![\phi]\!]$ denotes the set $\{s \in \Omega_D \mid [\phi]^s = \top\}$. We denote by RC (respectively RC_{bool}) the set of all relational calculus queries (respectively sentences). We consider only *domain-independent queries* and we employ the standard relational calculus semantics [2].

patient
Alice
Bob
Carl

smokes
Bob
Carl

father	
Bob	Carl

mother	
Alice	Carl

Fig. 2: The template for all database states, where the content of the *cancer* table is left unspecified.

An *integrity constraint over* D is a relational calculus sentence γ over D. Given a state s, we say that s *satisfies the constraint* γ iff $[\gamma]^s = \top$. Given a set of constraints Γ, Ω_D^Γ denotes the set of all states satisfying the constraints in Γ, i.e., $\Omega_D^\Gamma = \{s \in \Omega_D \mid \bigwedge_{\gamma \in \Gamma} [\gamma]^s = \top\}$.

Example IV.1. The database associated with the example in §II consists of five relational schemas *patient*, *smokes*, *cancer*, *father*, and *mother*, where the first three schemas have arity 1 and the last two have arity 2. We assume that there are only three patients `Alice`, `Bob`, and `Carl`, so the domain $\mathbf{dom}$ is $\{\texttt{Alice}, \texttt{Bob}, \texttt{Carl}\}$. The integrity constraints are as follows:

- `Alice`, `Bob`, and `Carl` are patients.

$$patient(\texttt{Alice}) \wedge patient(\texttt{Bob}) \wedge patient(\texttt{Carl})$$

- `Alice` and `Bob` are `Carl`'s parents.

$$\forall x, y.\ (father(x, y) \leftrightarrow (x = \texttt{Bob} \wedge y = \texttt{Carl})) \wedge$$
$$\forall x, y.\ (mother(x, y) \leftrightarrow (x = \texttt{Alice} \wedge y = \texttt{Carl}))$$

- `Alice` does not smoke, whereas `Bob` and `Carl` do.

$$\neg smokes(\texttt{Alice}) \wedge smokes(\texttt{Bob}) \wedge smokes(\texttt{Carl})$$

Given these constraints, there are just 8 possible database states in Ω_D^Γ, which differ only in their *cancer* relation. The content of the *cancer* relation is a subset of $\{\texttt{Alice}, \texttt{Bob}, \texttt{Carl}\}$, whereas the content of the other tables is shown in Figure 2. We denote each possible world as s_C, where the set $C \subseteq \{\texttt{Alice}, \texttt{Bob}, \texttt{Carl}\}$ denotes the users having cancer. ■

B. Security Policies

Existing access control models for databases are inadequate to formalize security requirements capturing probabilistic dependencies. For example, SQL cannot express statements like "A user u's belief that ϕ holds must be less than l." We present a simple framework, inspired by knowledge-based policies [52], for expressing such requirements.

A *D-secret* is a tuple $\langle U, \phi, l \rangle$, where U is either a finite set of users in $\mathcal{U}$ or a co-finite set of users, i.e., $U = \mathcal{U} \setminus U'$ for some finite $U' \subset \mathcal{U}$, ϕ is a relational calculus sentence over D, and l is rational number $0 \leq l \leq 1$ specifying the uncertainty threshold. Abusing notation, when U consists of a single user u, we write u instead of $\{u\}$. Informally, $\langle U, \phi, l \rangle$ represents that for each user $u \in U$, u's belief that ϕ holds in the actual database state must be less than l. Therefore, a command of the form `SECRET` q `FOR` u `THRESHOLD` l can be represented as $\langle u, q, l \rangle$, whereas a command `SECRET` q `FOR USERS NOT IN` $\{u_1, \ldots, u_n\}$ `THRESHOLD` l can be represented as $\langle \mathcal{U} \setminus \{u_1, \ldots, u_n\},$

$q, l\rangle$. Finally, a *D-security policy* is a finite set of D-secrets. Given a D-security policy P, we denote by $secrets(P, u)$ the set of D-secrets associated with the user u, i.e., $secrets(P, u) = \{\langle u, \phi, l\rangle \mid \langle U, \phi, l\rangle \in P \wedge u \in U\}$. Note that the function $secrets$ is computable since the set U is always either finite or co-finite.

Our framework also allows the specification of lower bounds. Requirements of the form "A user u's belief that ϕ holds must be greater than l" can be formalized as $\langle u, \neg\phi, 1-l\rangle$ (since the probability of $\neg\phi$ is $1-P(\phi)$, where $P(\phi)$ is ϕ's probability). Security policies can be extended to support secrets over non-boolean queries. A secret $\langle u, \{\overline{x} \mid \phi(\overline{x})\}, l\rangle$ can be seen as a shorthand for the set $\{\langle u, \phi[\overline{x} \mapsto \overline{t}], l\rangle \mid \overline{t} \in \bigcup_{s\in\Omega_D^\Gamma}[\{\overline{x} \mid \phi(\overline{x})\}]^s\}$, i.e., u's belief in any tuple $\overline{t}$ being in the query's result must be less than l.

Example IV.2. Let *Mallory* denote the malicious researcher from §II and D be the schema from Example IV.1. Consider the requirement from §II: *Mallory*'s belief in a patient having cancer must be less than 50%. This can be formalized as $\langle Mallory, cancer(\texttt{Alice}), 1/2\rangle$, $\langle Mallory, cancer(\texttt{Bob}), 1/2\rangle$, and $\langle Mallory, cancer(\texttt{Carl}), 1/2\rangle$, or equivalently as $\langle Mallory, \{p \mid cancer(p)\}, 1/2\rangle$. In contrast, the requirement "For all users u that are not *Carl*, u's belief in *Carl* having cancer must be less than 50%" can be formalized as $\langle \mathcal{U} \setminus \{Carl\}, cancer(\texttt{Carl}), 1/2\rangle$, where *Carl* denotes the user identifier associated with Carl. ■

C. Formalized System Model

A *system configuration* is a tuple $\langle D, \Gamma\rangle$, where D is a database schema and Γ is a set of D-integrity constraints. Let $C = \langle D, \Gamma\rangle$ be a system configuration. A *C-system state* is a tuple $\langle db, U, P\rangle$, where $db \in \Omega_D^\Gamma$ is a database state, $U \subset \mathcal{U}$ is a finite set of users, and P is a D-security policy. A *C-query* is a pair $\langle u, \phi\rangle$ where $u \in \mathcal{U}$ is a user and ϕ is a relational calculus sentence over D.[2] We denote by Ω_C the set of all system states and by $\mathcal{Q}_C$ the set of all queries.

A *C-event* is a triple $\langle q, a, res\rangle$, where q is a C-query in $\mathcal{Q}_C$, $a \in \{\top, \bot\}$ is a security decision, where $\top$ stands for "authorized query" and $\bot$ stands for "unauthorized query", and $res \in \{\top, \bot, \dagger\}$ is the query's result, where $\top$ and $\bot$ represent the usual boolean values and $\dagger$ represents that the query was not executed as access was denied. Given a C-event $e = \langle q, a, res\rangle$, we denote by $q(e)$ (respectively $a(e)$ and $res(e)$) the query q (respectively the decision a and the result res). A *C-history* is a finite sequence of C-events. We denote by $\mathcal{H}_C$ the set of all possible C-histories. Moreover, given a sequence h, $|h|$ denotes its length, $h(i)$ its i-th element, and h^i the sequence containing the first i elements of h. We also denote by h^0 the empty sequence ϵ, and $\cdot$ denotes the concatenation operator.

[2]Without loss of generality, we focus only on boolean queries [2]. We can support non-boolean queries as follows. Given a database state s and a query $q := \{\overline{x} \mid \phi\}$, if the inference control mechanism authorizes the boolean query $\bigwedge_{\overline{t}\in[q]^s} \phi[\overline{x} \mapsto \overline{t}] \wedge (\forall \overline{x}.\, \phi \rightarrow \bigvee_{\overline{t}\in[q]^s} \overline{x} = \overline{t})$, then we return q's result, and otherwise we reject q as unauthorized.

We now formalize Policy Decision Points. A *C-PDP* is a function $f : \Omega_C \times \mathcal{Q}_C \times \mathcal{H}_C \rightarrow \{\top, \bot\}$ taking as input a system state, a query, and a history and returning the security decision, accept ($\top$) or deny ($\bot$).

Let C be a system configuration, $s = \langle db, U, P\rangle$ be a C-state, and f be a C-PDP. A C-history h is *compatible with s and f* iff for each $1 \leq i \leq |h|$, (1) $f(s, q(h(i)), h^{i-1}) = a(h(i))$, (2) if $a(h(i)) = \bot$, then $res(h(i)) = \dagger$, and (3) if $a(h(i)) = \top$, then $res(h(i)) = [\phi]^{db}$, where $q(h(i)) = \langle u, \phi\rangle$. In other words, h is compatible with s and f iff it was generated by the PDP f starting in state s.

A *(C, f)-run* is a pair $\langle s, h\rangle$, where s is a system state in Ω_C and h is a history in $\mathcal{H}_C$ compatible with s and f. Since all queries are `SELECT` queries, the system state does not change along the run. Hence, our runs consist of a state and a history instead of e.g., an alternating sequence of states and actions (as is standard for runs). We denote by $runs(C, f)$ the set of all (C, f)-runs. Furthermore, given a run $r = \langle\langle db, U, P\rangle, h\rangle$, we denote by r^i the run $\langle\langle db, U, P\rangle, h^i\rangle$, and we use dot notation to access to r's components. For instance, $r.db$ denotes the database state db and $r.h$ denotes the history.

Example IV.3. Consider the run $r = \langle\langle db, U, P\rangle, h\rangle$, where the database state db is the state $s_{\{\texttt{A},\texttt{B},\texttt{C}\}}$, where `Alice`, `Bob`, and `Carl` have cancer, the policy P is defined in Example IV.2, the set of users U contains only *Mallory*, and the history h is as follows (here we assume that all queries are authorized):

1) *Mallory* checks whether *Carl* smokes. Thus, $h(1) = \langle\langle Mallory, smokes(\texttt{Carl})\rangle, \top, \top\rangle$.
2) *Mallory* checks whether *Carl* is *Alice*'s and *Bob*'s son. Therefore, $h(2)$ is $\langle\langle Mallory, father(\texttt{Bob}, \texttt{Carl}) \wedge mother(\texttt{Alice}, \texttt{Carl})\rangle, \top, \top\rangle$.
3) *Mallory* checks whether *Alice* has cancer. Thus, $h(3) = \langle\langle Mallory, cancer(\texttt{Alice})\rangle, \top, \top\rangle$.
4) *Mallory* checks whether *Bob* has cancer. Thus, $h(4) = \langle\langle Mallory, cancer(\texttt{Bob})\rangle, \top, \top\rangle$. ■

D. Attacker Model

To reason about DBIC, it is essential to precisely define (1) how users interact with the system, (2) how they reason about the system's behaviour, (3) their initial beliefs about the database state, and (4) how these beliefs change by observing the system's behaviour. We formalize this in an attacker model.

Each user has an initial belief about the database state. Following [18], [19], [29], [52], we represent a user's beliefs as a probability distribution over all database states. Furthermore, users observe the system's behaviour and derive information about the database content. We formalize a user's observations as an equivalence relation over runs, where two runs are equivalent iff the user's observations are the same in both runs, as is standard in information-flow [7], [8]. A user's knowledge is the set of all database states that he considers possible given his observations. Finally, we use Bayesian conditioning to update a user's beliefs given his knowledge.

Let $C = \langle D, \Gamma\rangle$ be a system configuration and f be a C-PDP. A *C-probability distribution* is a discrete probability

distribution given by a function $P : \Omega_D^\Gamma \to [0,1]$ such that $\sum_{db \in \Omega_D^\Gamma} P(db) = 1$. Given a set $E \subseteq \Omega_D^\Gamma$, $P(E)$ denotes $\sum_{s \in E} P(s)$. Furthermore, given two sets $E', E'' \subseteq \Omega_D^\Gamma$ such that $P(E') \neq 0$, $P(E'' \mid E')$ denotes $P(E'' \cap E')/P(E')$ as is standard. We denote by $\mathcal{P}_C$ the set of all possible C-probability distributions. Abusing notation, we extend probability distributions to formulae: $P(\psi) = P(\llbracket \psi \rrbracket)$, where $\llbracket \psi \rrbracket = \{ db \in \Omega_D^\Gamma \mid [\psi]^{db} = \top \}$.

We now introduce *indistinstinguishability*, an equivalence relation used in information-flow control [42]. Let C be a system configuration and f be a C-PDP. Given a history h and a user $u \in \mathcal{U}$, $h|_u$ denotes the history obtained from h by removing all C-events from users other than u, namely $\epsilon|_u = \epsilon$, and if $h = \langle\langle u', q\rangle, a, res\rangle \cdot h'$, then $h|_u = h'|_u$ in case $u \neq u'$, and $h|_u = \langle\langle u, q\rangle, a, res\rangle \cdot h'|_u$ if $u = u'$. Given two runs $r = \langle\langle db, U, P\rangle, h\rangle$ and $r' = \langle\langle db', U', P'\rangle, h'\rangle$ in $runs(C, f)$ and a user $u \in \mathcal{U}$, we say that r and r' are *indistinguishable for* u, written $r \sim_u r'$, iff $h|_u = h'|_u$. This means that r and r' are indistinguishable for a user u iff the system's behaviour in response to u's commands is the same in both runs. Note that $\sim_u$ depends on both C and f, which we generally leave implicit. Given a run r, $[r]_{\sim_u}$ is the equivalence class of r with respect to $\sim_u$, i.e, $[r]_{\sim_u} = \{r' \in runs(C, f) \mid r' \sim_u r\}$, whereas $\llbracket r \rrbracket_{\sim_u}$ is set of all databases associated to the runs in $[r]_{\sim_u}$, i.e., $\llbracket r \rrbracket_{\sim_u} = \{ db \mid \exists U, P, h.\ \langle\langle db, U, P\rangle, h\rangle \in [r]_{\sim_u} \}$.

Definition IV.1. Let $C = \langle D, \Gamma\rangle$ be a configuration and f be a C-PDP. A *(C, f)-attacker model* is a function $ATK : \mathcal{U} \to \mathcal{P}_C$ associating to each user $u \in \mathcal{U}$ a C-probability distribution representing u's initial beliefs. Additionally, for all users $u \in \mathcal{U}$ and all states $s \in \Omega_D^\Gamma$, we require that $ATK(u)(s) > 0$. The *semantics of* ATK is $\llbracket ATK \rrbracket(u, r) = \lambda s \in \Omega_D^\Gamma.\ ATK(u)(s \mid \llbracket r \rrbracket_{\sim_u})$, where $u \in \mathcal{U}$ and $r \in runs(C, f)$. □

The semantics of an attacker model ATK associates to each user u and each run r the probability distribution obtained by updating u's initial beliefs given his knowledge with respect to the run r. We informally refer to $\llbracket ATK \rrbracket(u, r)(\llbracket \phi \rrbracket)$ as u's beliefs in a sentence ϕ (given a run r).

Example IV.4. The attacker model for the example from §II is as follows. Let $X_{\texttt{Alice}}$, $X_{\texttt{Bob}}$, and $X_{\texttt{Carl}}$ be three boolean random variables, representing the probability that the corresponding patient has cancer. They define the following joint probability distribution, which represents a user's initial beliefs about the actual database state: $P(X_{\texttt{Alice}}, X_{\texttt{Bob}}, X_{\texttt{Carl}}) = P(X_{\texttt{Alice}}) \cdot P(X_{\texttt{Bob}}) \cdot P(X_{\texttt{Carl}} \mid X_{\texttt{Alice}}, X_{\texttt{Bob}})$. The probability distributions of these variables are given in Figure 3 and they are derived from the probabilistic model in §II. We associate each outcome (x, y, z) of $X_{\texttt{Alice}}, X_{\texttt{Bob}}, X_{\texttt{Carl}}$ with the corresponding database state s_C, where C is the set of patients such that the outcome of the corresponding variable is $\top$. For each user $u \in \mathcal{U}$, the distribution P_u is defined as $P_u(s_C) = P(X_{\texttt{Alice}} = x, X_{\texttt{Bob}} = y, X_{\texttt{Carl}} = z)$, where x (respectively y and z) is $\top$ if `Alice` (respectively `Bob` and `Carl`) is in C and $\bot$ otherwise. Figure 4 shows the probabilities associated with each state in Ω_D^Γ, i.e., a user's initial beliefs. Finally, the attacker model is $ATK = \lambda u \in \mathcal{U}.P_u$. ■

	$X_{\texttt{Alice}}$
$\top$	$1/20$
$\bot$	$19/20$

	$X_{\texttt{Bob}}$
$\top$	$6/20$
$\bot$	$14/20$

$X_{\texttt{Alice}}$	$X_{\texttt{Bob}}$	$X_{\texttt{Carl}}$ $\top$	$X_{\texttt{Carl}}$ $\bot$
$\top$	$\top$	$12/20$	$8/20$
$\top$	$\bot$	$9/20$	$11/20$
$\bot$	$\top$	$9/20$	$11/20$
$\bot$	$\bot$	$6/20$	$14/20$

Fig. 3: Probability distribution for the random variables $X_{\texttt{Alice}}$, $X_{\texttt{Bob}}$, and $X_{\texttt{Carl}}$ from Example IV.4.

State	Probability	State	Probability
$s_\emptyset$	0.4655	$s_{\{\texttt{A,B}\}}$	0.006
$s_{\{\texttt{A}\}}$	0.01925	$s_{\{\texttt{A,C}\}}$	0.01575
$s_{\{\texttt{B}\}}$	0.15675	$s_{\{\texttt{B,C}\}}$	0.12825
$s_{\{\texttt{C}\}}$	0.1995	$s_{\{\texttt{A,B,C}\}}$	0.009

Fig. 4: Probability distribution over all database states. Each state is denoted as s_C, where C is the content of the *cancer* table. Here we denote the patients' names with their initials.

E. Confidentiality

We first define the notion of a secrecy-preserving run for a secret $\langle u, \phi, l\rangle$ and an attacker model ATK. Informally, a run r is secrecy-preserving for $\langle u, \phi, l\rangle$ iff whenever an attacker's belief in the secret ϕ is below the threshold l, then there is no way for the attacker to increase his belief in ϕ above the threshold. Our notion of secrecy-preserving runs is inspired by existing security notions for query auditing [29].

Definition IV.2. Let $C = \langle D, \Gamma\rangle$ be a configuration, f be a C-PDP, and ATK be a (C, f)-attacker model. A run r is *secrecy-preserving* for a secret $\langle u, \phi, l\rangle$ and ATK iff for all $0 \leq i < |r|$, $\llbracket ATK \rrbracket(u, r^i)(\phi) < l$ implies $\llbracket ATK \rrbracket(u, r^{i+1})(\phi) < l$. □

We now formalize our confidentiality notion. A PDP provides data confidentiality for an attacker model ATK iff all runs are secrecy-preserving for ATK. Note that our security notion can be seen as a probabilistic generalization of opacity [60] for the database setting. Our notion is also inspired by the semantics of knowledge-based policies [52].

Definition IV.3. Let $C = \langle D, \Gamma\rangle$ be a system configuration, f be a C-PDP, and ATK be a (C, f)-attacker model. We say that the PDP f *provides data confidentiality with respect to C and ATK* iff for all runs $r = \langle\langle db, U, P\rangle, h\rangle$ in $runs(C, f)$, for all users $u \in U$, for all secrets $s \in secrets(P, u)$, r is secrecy-preserving for s and ATK. □

A PDP providing confidentiality ensures that if an attacker's initial belief in a secret ϕ is below the corresponding threshold, then there is no way for the attacker to increase his belief in ϕ above the threshold by interacting with the system. This guarantee does not however apply to *trivial non-secrets*, i.e., those secrets an attacker knows with a probability at least the threshold even before interacting with the system. No PDP can prevent their disclosure since the disclosure does not depend on the attacker's interaction with the database.

Example IV.5. Let r be the run given in Example IV.3, ATK be the attacker model in Example IV.4, and u be the user *Mallory*. In the following, ϕ_1, ϕ_2, and ϕ_3 denote $cancer(\texttt{Carl})$, $cancer(\texttt{Bob})$, and $cancer(\texttt{Alice})$ respectively, i.e., the three secrets from Example IV.2. Furthermore, we assume that the policy contains an additional secret $\langle Mallory, \phi_4, 1/2\rangle$, where $\phi_4 := \neg cancer(\texttt{Alice})$.

Figure 5 illustrates *Mallory*'s beliefs about $\phi_1, \ldots, \phi_4$ and whether the run is secrecy-preserving for the secrets $\phi_1, \ldots, \phi_4$. The probabilities in the tables can be obtained by combining the states in $[\![r^i]\!]_{\sim_u}$, for $0 \leq i \leq 4$, and $[\![\phi_j]\!]$, for $1 \leq j \leq 4$, with the probabilities from Figure 4. As shown in Figure 5, the run is not secrecy-preserving for the secrets ϕ_1 and ϕ_2 as it completely discloses that *Alice* and *Bob* have cancer, in the third and fourth steps respectively. Secrecy-preservation is also violated for the secret ϕ_1, even though r does not directly disclose any information about *Carl*'s health status. Indeed, in the last step of the run, *Mallory*'s belief in ϕ_1 is 0.6, which is higher than the threshold $1/2$, even though his belief in ϕ_1 before learning that *Bob* had cancer was below the threshold. Note that ϕ_4 is a trivial non-secret: even before interacting with the system, *Mallory*'s belief in ϕ_4 is 0.95. ■

F. Discussion

Our approach assumes that the attacker's capabilities are well-defined. While this, in general, is a strong assumption, there are many domains where such information is known. There are, however, domains where this information is lacking. In these cases, security engineers must (1) determine the appropriate beliefs capturing the desired attacker models, and (2) formalize them. The latter can be done, for instance, using ATKLOG (see §V). Note however that precisely eliciting the attackers' capabilities is still an open problem in DBIC.

V. ATKLOG

A. Probabilistic Logic Programming

PROBLOG [20], [21], [31] is a probabilistic logic programming language with associated tool support. An exact inference engine for PROBLOG is available at [1].

Conventional logic programs are constructed from terms, atoms, literals, and rules. In the following, we consider only function-free logic programs, also called DATALOG programs. In this setting, terms are either variable identifiers or constants.

Let Σ be a first-order signature, $\mathbf{dom}$ be a finite domain, and Var be a countably infinite set of variable identifiers. A $(\Sigma, \mathbf{dom})$-*atom* $R(v_1, \ldots, v_n)$ consists of a predicate symbol $R \in \Sigma$ and arguments $v_1, \ldots, v_n$ such that n is the arity of R, and each v_i, for $1 \leq i \leq n$, is either a variable identifier in Var or a constant in $\mathbf{dom}$. We denote by $\mathcal{A}_{\Sigma,\mathbf{dom}}$ the set $\{R(v_1, \ldots, v_{|R|}) \mid R \in \Sigma \wedge v_1, \ldots, v_{|R|} \in \mathbf{dom} \cup Var\}$ of all $(\Sigma, \mathbf{dom})$-atoms. A $(\Sigma, \mathbf{dom})$-*literal* l is either a $(\Sigma, \mathbf{dom})$-atom a or its negation $\neg a$, where $a \in \mathcal{A}_{\Sigma,\mathbf{dom}}$. We denote by $\mathcal{L}_{\Sigma,\mathbf{dom}}$ the set $\mathcal{A}_{\Sigma,\mathbf{dom}} \cup \{\neg a \mid a \in \mathcal{A}_{\Sigma,\mathbf{dom}}\}$ of all $(\Sigma, \mathbf{dom})$-literals. Given a literal l, $vars(l)$ denotes the set of its variables, $args(l)$ the list of its arguments, and $pred(l)$ the predicate symbol used in l. As is standard, we say that a literal l is *positive* if it is an atom in $\mathcal{A}_{\Sigma,\mathbf{dom}}$ and *negative* if it is the negation of an atom. Furthermore, we say that a literal l is *ground* iff $vars(l) = \emptyset$.

A $(\Sigma, \mathbf{dom})$-*rule* is of the form $h \leftarrow l_1, \ldots, l_n, e_1, \ldots, e_m$, where $h \in \mathcal{A}_{\Sigma,\mathbf{dom}}$ is a $(\Sigma, \mathbf{dom})$-atom, $l_1, \ldots, l_n \in \mathcal{L}_{\Sigma,\mathbf{dom}}$ are $(\Sigma, \mathbf{dom})$-literals, and $e_1, \ldots, e_m$ are equality and inequality constraints over the variables in $h, l_1, \ldots, l_m$.[3] Given a rule r, we denote by $head(r)$ the atom h, by $body(r)$ the literals $l_1, \ldots, l_n$, by $cstr(r)$ the constraints $e_1, \ldots, e_m$, and by $body(r, i)$ the i-th literal in r's body, i.e., $body(r, i) = l_i$. Furthermore, we denote by $body^+(r)$ (respectively $body^-(r)$) all positive (respectively negative) literals in $body(r)$. As is standard, we assume that the free variables in a rule's head are a subset of the free variables of the positive literals in the rule's body, i.e., $vars(head(r)) \subseteq \bigcup_{l \in body^+(r)} vars(l) \cup \bigcup_{(x=c) \in cstr(r) \wedge c \in \mathbf{dom}} \{x\}$. Finally, a $(\Sigma, \mathbf{dom})$-*logic program* is a set of $(\Sigma, \mathbf{dom})$-ground atoms and $(\Sigma, \mathbf{dom})$-rules. We consider only programs p that do not contain negative cycles in the rules as is standard for stratified DATALOG [2].

To reason about probabilities, PROBLOG extends logic programming with probabilistic atoms. A $(\Sigma, \mathbf{dom})$-*probabilistic atom* is a $(\Sigma, \mathbf{dom})$-atom a annotated with a value $0 \leq v \leq 1$, denoted $v::a$. PROBLOG supports both probabilistic ground atoms and rules having probabilistic atoms in their heads. PROBLOG also supports *annotated disjunctions* $v_1::a_1; \ldots; v_n::a_n$, where $a_1, \ldots, a_n$ are ground atoms and $\left(\sum_{1 \leq i \leq n} v_i\right) \leq 1$, which denote that $a_1, \ldots, a_n$ are mutually exclusive probabilistic events happening with probabilities $v_1, \ldots, v_n$. Annotated disjunctions can either be used as ground atoms or as heads in rules. Both annotated disjunctions and probabilistic rules are just syntactic sugar and can be expressed using ground probabilistic atoms and standard rules [20], [21], [31]; see Appendix A.

A $(\Sigma, \mathbf{dom})$-PROBLOG program p defines a probability distribution over all possible $(\Sigma, \mathbf{dom})$-structures, denoted $[\![p]\!]$. Note that we consider only function-free PROBLOG programs. Hence, in our setting, PROBLOG is a probabilistic extension of DATALOG. Appendix A contains a formal account of PROBLOG's semantics.

Medical Data. We formalize the probability distribution from Example IV.4 as a PROBLOG program. We reuse the database schema and the domain from Example IV.1 as the first-order signature and the domain for the PROBLOG program. First, we encode the template shown in Figure 2 using ground atoms: $patient(\texttt{Alice})$, $patient(\texttt{Bob})$, $patient(\texttt{Carl})$, $smokes(\texttt{Bob})$, $smokes(\texttt{Carl})$, $father(\texttt{Bob}, \texttt{Carl})$, and $mother(\texttt{Alice}, \texttt{Carl})$. Second, we encode the probability distribution associated with the possible values of the *cancer* table using the following PROBLOG rules, which have probabilistic atoms in their heads:

$$1/20::cancer(x) \leftarrow patient(x)$$
$$5/19::cancer(x) \leftarrow smokes(x)$$

[3] Without loss of generality, we assume that equality constraints involving a variable v and a constant c are of the form $v = c$.

i	$[\![ATK]\!](u, r^i)([\![\phi]\!])$				$[\![ATK]\!](u, r^{i+1})([\![\phi]\!])$				Secrecy			
	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_1	ϕ_2	ϕ_3	ϕ_4
0	0.3525	0.3	0.05	0.95	0.3525	0.3	0.05	0.95	✓	✓	✓	$*$
1	0.3525	0.3	0.05	0.95	0.3525	0.3	0.05	0.95	✓	✓	✓	$*$
2	0.3525	0.3	0.05	0.95	0.495	0.3	1	0	✓	✓	$\mathcal{X}$	$*$
3	0.495	0.3	1	0	0.6	1	1	0	$\mathcal{X}$	$\mathcal{X}$	$\mathcal{X}$	$*$
4	0.6	1	1	0	–	–	–	–	–	–	–	–

Fig. 5: Evolution of *Mallory*'s beliefs in the secrets $\phi_1, \ldots, \phi_4$ for the run r and the attacker model ATK from Example IV.5. In the table, $\mathcal{X}$ and ✓ denote that secrecy-preservation is violated and satisfied respectively, whereas $*$ denotes trivial secrets.

$$
\begin{aligned}
{}^{3}/_{14}{::}\mathit{cancer}(y) \leftarrow\ & \mathit{father}(x,y), \mathit{cancer}(x),\\
& \mathit{mother}(z,y), \neg\mathit{cancer}(z)\\
{}^{3}/_{14}{::}\mathit{cancer}(y) \leftarrow\ & \mathit{father}(x,y), \neg\mathit{cancer}(x),\\
& \mathit{mother}(z,y), \mathit{cancer}(z)\\
{}^{3}/_{7}{::}\mathit{cancer}(y) \leftarrow\ & \mathit{father}(x,y), \mathit{cancer}(x),\\
& \mathit{mother}(z,y), \mathit{cancer}(z)
\end{aligned}
$$

The coefficients in the above example are derived from §II. For instance, the probability that a smoking patient x whose parents are not not in the *cancer* relation has cancer is 30%. The coefficient in the first rule is ${}^{1}/_{20}$ since each patient has a 5% probability of having cancer. The coefficient in the second rule is ${}^{5}/_{19}$, which is $({}^{6}/_{20} - {}^{1}/_{20}) \cdot (1 - {}^{1}/_{20})^{-1}$, i.e., the probability that $\mathit{cancer}(x)$ is derived from the second rule given that it has not been derived from the first rule. This ensures that the overall probability of deriving $\mathit{cancer}(x)$ is ${}^{6}/_{20}$, i.e., 30%. The coefficients for the last two rules are derived analogously.

Informally, a probabilistic ground atom ${}^{1}/_{2}{::}\mathit{cancer}(\texttt{Bob})$ expresses that $\mathit{cancer}(\texttt{Bob})$ holds with a probability ${}^{1}/_{2}$. Similarly, the rule ${}^{1}/_{20}{::}\mathit{cancer}(x) \leftarrow \mathit{patient}(x)$ states that, for any x such that $\mathit{patient}(x)$ holds, then $\mathit{cancer}(x)$ can be derived with probability ${}^{1}/_{20}$. This program yields the probability distribution shown in Figure 4.

B. AtkLog's Foundations

We first introduce belief programs, which formalize an attacker's initial beliefs. Afterwards, we formalize AtkLog.

Belief Programs. A belief program formalizes an attacker's beliefs as a probability distribution over the database states.

A database schema $D' = \langle \Sigma', \mathbf{dom} \rangle$ extends a schema $D = \langle \Sigma, \mathbf{dom} \rangle$ iff Σ' contains all relation schemas in Σ. The extension D' may extend Σ with additional predicate symbols necessary to encode probabilistic dependencies. Given an extension D', a D'-state s' *agrees* with a D-state s iff $s'(R) = s(R)$ for all R in D. Given a D-state s, we denote by $EXT(s, D, D')$ the set of all D'-states that agree with s.

A $(\Sigma', \mathbf{dom})$-ProbLog program p, where $D' = \langle \Sigma', \mathbf{dom} \rangle$ extends D, is a *belief program over* D. The *D-semantics of* p is $[\![p]\!]_D = \lambda s \in \Omega_D.\ \sum_{s' \in EXT(s,D,D')} [\![p]\!](s')$. Given a system configuration $C = \langle D, \Gamma \rangle$, a belief program p over D *complies with* C iff $[\![p]\!]_D$ is a C-probability distribution. With a slight abuse of notation, we lift the semantics of belief programs to sentences: $[\![p]\!]_D = \lambda\phi \in \mathcal{RC}_{bool}.\ \sum_{s' \in \{s \in \Omega_D \mid [\phi]^s = \top\}} [\![p]\!]_D(s')$.

AtkLog. An AtkLog model specifies the initial beliefs of all users in $\mathcal{U}$ using belief programs.

Let D be a database schema and $C = \langle D, \Gamma \rangle$ be a system configuration. A C-AtkLog *model* ATK is a function associating to each user $u \in U$, where $U \subset \mathcal{U}$ is a finite set of users, a belief program p_u and to all users $u \in \mathcal{U} \setminus U$ a belief program p_0, such that for all users $u \in \mathcal{U}$, $[\![ATK(u)]\!]_D$ complies with C and for all database states $s \in \Omega_D^\Gamma$, $[\![ATK(u)]\!]_D(s) > 0$, i.e., all database states satisfying the integrity constraints are possible. Informally, a C-AtkLog model associates a distinct belief program to each user in U, and it associates to each user in $\mathcal{U} \setminus U$ the same belief program p_0.

Given a C-PDP f, a C-AtkLog model ATK defines the (C, f)-attacker model $\lambda u \in \mathcal{U}.[\![ATK(u)]\!]_D$ that associates to each user $u \in \mathcal{U}$ the probability distribution defined by the belief program $ATK(u)$. The semantics of this (C, f)-attacker model is: $\lambda u \in \mathcal{U}.\lambda r \in \mathit{runs}(C, f).\lambda s \in \Omega_D^\Gamma.\ [\![ATK(u)]\!]_D(s \mid [\![r]\!]_{\sim_u})$. Informally, given a C-AtkLog model ATK, a C-PDP f, and a user u, u's belief in a database state s, given a run r, is obtained by conditioning the probability distribution defined by the belief program $ATK(u)$ given the set of database states corresponding to all runs $r' \sim_u r$.

VI. Tractable Inference for ProbLog programs

Probabilistic inference in ProbLog is intractable in general. Its data complexity, i.e., the complexity of inference when only the programs' probabilistic ground atoms are part of the input and the rules are considered fixed and not part of the input, is $\#P$-hard; see [36]. This limits the practical applicability of ProbLog (and AtkLog) for DBIC. To address this, we define acyclic ProbLog programs, a class of programs where the data complexity of inference is PTime.

Given a ProbLog program p, our inference algorithm consists of three steps: (1) we compute all of p's derivations, (2) we compile these derivations into a Bayesian Network (BN) bn, and (3) we perform the inference over bn. To ensure tractability, we leverage two key insights. First, we exploit guarded negation [9] to develop a sound over-approximation, called the relaxed grounding, of all derivations of a program that is independent of the presence (or absence) of the probabilistic atoms. This ensures that whenever a ground atom can be derived from a program (for a possible assignment to the probabilistic atoms), the atom is also part of this program's relaxed grounding. This avoids grounding p for each possible assignment to the probabilistic atoms. Second, we introduce syntactic constraints (acyclicity) that ensure that

bn is a forest of poly-trees. This ensures tractability since inference for poly-tree BNs can be performed in polynomial time in the network's size [48].

We also precisely characterize the expressiveness of acyclic ProbLog programs. In this respect, we prove that acyclic programs are as expressive as forests of poly-tree BNs, one of the few classes of BNs with tractable inference.

As mentioned in §V, probabilistic rules and annotated disjunctions are just syntactic sugar. Hence, in the following we consider ProbLog programs consisting just of probabilistic ground atoms and non-probabilistic rules. Note also that we treat ground atoms as rules with an empty body.

A. Preliminaries

Negation-guarded Programs. A rule r is *negation-guarded* [9] iff all the variables occurring in negative literals also occur in positive literals, namely for all negative literals l in $body^{-}(r)$, $vars(l) \subseteq \bigcup_{l' \in body^{+}(r)} vars(l')$. To illustrate, the rule $C(x) \leftarrow A(x), \neg B(x)$ is negation-guarded, whereas $C(x) \leftarrow A(x), \neg B(x, y)$ is not since the variable y does not occur in any positive literal. We say that a program p is *negation-guarded* if all rules $r \in p$ are.

Relaxed Grounding. The relaxed grounding of a program p is obtained by considering all probabilistic atoms as certain and by grounding all positive literals. For all negation-guarded programs, the relaxed grounding of p is a sound over-approximation of all possible derivations in p. Given a program p and a rule $r \in p$, $rg(p)$ denotes p's relaxed grounding and $rg(p, r)$ denotes the set of r's ground instances. We formalize relaxed groundings in [36].

Example VI.1. Let p be the program consisting of the facts $1/2::A(1)$, $A(2)$, $A(3)$, $D(1)$, $E(2)$, $F(1)$, $O(1, 2)$, and $2/3::O(2, 3)$, and the rules $r_a = B(x) \leftarrow A(x), D(x)$, $r_b = B(x) \leftarrow A(x), E(x)$, and $r_c = B(y) \leftarrow B(x), \neg F(x), O(x, y)$. The relaxed grounding of p consists of the initial facts together with $B(1)$, $B(2)$, and $B(3)$, whereas $rg(p, r_c)$ consists of $B(2) \leftarrow B(1), \neg F(1), O(1, 2)$ and $B(3) \leftarrow B(2), \neg F(2), O(2, 3)$. ■

Dependency and Ground Graphs. The *dependency graph* of a program p, denoted $graph(p)$, is the directed labelled graph having as nodes all the predicate symbols in p and having an edge $a \xrightarrow{r,i} b$ iff there is a rule r such that a occurs in i-th literal in r's body and b occurs in r's head. Figure 6 depicts the dependency graph from Example VI.1. The *ground graph* of a program p is the graph obtained from its relaxed grounding. Hence, there is an edge $a \xrightarrow{r,gr,i} b$ from the ground atom a to the ground atom b iff there is a rule r and a ground rule $gr \in rg(p, r)$ such that $body(gr, i) \in \{a, \neg a\}$ and $head(gr) = b$. Figure 7 depicts the ground graph from Example VI.1. Note that there are no incoming or outgoing edges from $A(3)$ because the node is not involved in any derivation.

Propagation Maps. We use propagation maps to track how information flows inside rules. Given a rule r and a literal $l \in body(r)$, the (r, l)-*vertical map* is the mapping μ from $\{1, \ldots, |l|\}$ to $\{1, \ldots, |head(r)|\}$ such that $\mu(i) = j$ iff $args(l)(i) = args(head(r))(j)$ and $args(l)(i) \in Var$. Given a rule r and literals l and l' in r's body, the (r, l, l')-*horizontal map* is the mapping μ from $\{1, \ldots, |l|\}$ to $\{1, \ldots, |l'|\}$ such that $\mu(i) = j$ iff $args(l)(i) = args(l')(j)$ and $args(l)(i) \in Var$.

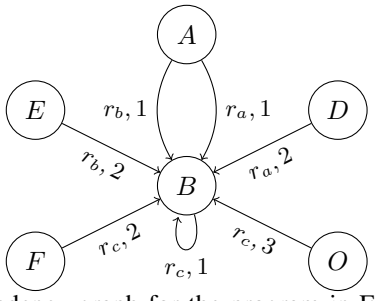


Fig. 6: Dependency graph for the program in Example VI.1.

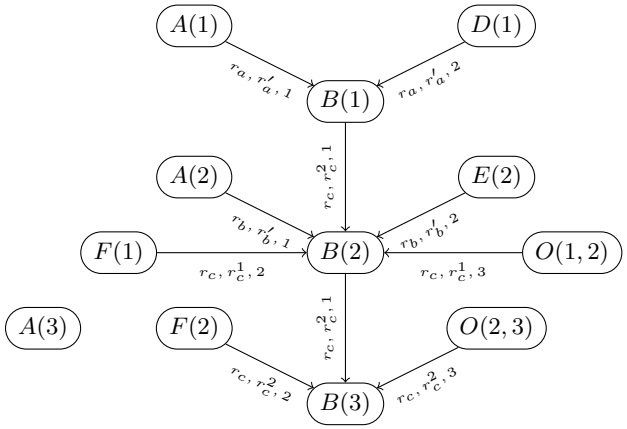


Fig. 7: Ground graph for the program in Example VI.1. The ground rules r'_a, r'_b, r^1_c, and r^2_c are as follows: $r'_a = B(1) \leftarrow A(1), D(1)$, $r'_b = B(2) \leftarrow A(2), E(2)$, $r^1_c = B(2) \leftarrow B(1), \neg F(1), O(1, 2)$, and $r^2_c = B(3) \leftarrow B(2), \neg F(2), O(2, 3)$.

We say that a path links to a literal l if information flows along the rules to l. This can be formalized by posing constraints on the mapping obtained by combining horizontal and vertical maps along the path. Formally, given a literal l and a mapping $\nu : \mathbb{N} \to \mathbb{N}$, a directed path $pr_1 \xrightarrow{r_1,i_1} \ldots \xrightarrow{r_{n-1},i_{n-1}} pr_n$ *ν-downward links to l* iff there is a $0 \leq j < n - 1$ such that the function $\mu := \mu' \circ \mu_j \circ \ldots \circ \mu_1$ satisfies $\mu(k) = \nu(k)$ for all k for which $\nu(k)$ is defined, where for $1 \leq h \leq j$, μ_h is the $(r_h, body(r_h, i_h))$-vertical map, and μ' is the horizontal map connecting $body(r_{j+1}, i_{j+1})$ with l. Similarly, a directed path $pr_1 \xrightarrow{r_1,i_1} \ldots \xrightarrow{r_{n-1},i_{n-1}} pr_n$ *ν-upward links to l* iff there is a $1 \leq j \leq n - 1$ such that the function $\mu := \mu'^{-1} \circ \mu^{-1}_{j+1} \circ \ldots \circ \mu^{-1}_{n-1}$ satisfies $\mu(k) = \nu(k)$ for all k for which $\nu(k)$ is defined, where μ_h is the $(r_h, body(r_h, i_h))$-vertical map, for $j < h \leq n - 1$, and μ' is the (r_j, l)-vertical map. A path P *links to a predicate symbol a* iff there is an atom $a(\overline{x})$ such that P links to $a(\overline{x})$.

Example VI.2. The horizontal map connecting $A(x)$ and $D(x)$ in r_a, i.e., the $(r_a, A(x), D(x))$-horizontal map, is

$\{1 \mapsto 1\}$. The horizontal map connecting $A(x)$ and $E(x)$ in r_b is $\{1 \mapsto 1\}$ as well. Hence, the path $A \xrightarrow{r_a,1} B$ downward links to D and the path $A \xrightarrow{r_b,1} B$ downward links to E for the mapping $\{1 \mapsto 1\}$. Furthermore, the path $B \xrightarrow{r_c,1} B$ downward links to O for $\{1 \mapsto 1\}$ since the $(r_c, B(x), O(x, y))$-horizontal map is $\{1 \mapsto 1\}$. Finally, the path $B \xrightarrow{r_c,1} B$ upward links to O for $\{2 \mapsto 1\}$ since the $(r_c, O(x, y))$-vertical map is $\{2 \mapsto 1\}$. ■

B. Acyclic ProbLog programs

A sufficient condition for tractable inference is that p's ground graph is a forest of poly-trees. This requires that p's ground graph neither contains directed nor undirected cycles, or, equivalently, the undirected version of p's ground graph is acyclic. To illustrate, the ground graph in Figure 7 is a poly-tree. The key insight here is that a cycle among ground atoms is caused by a (directed or undirected) cycle among p's predicate symbols. In a nutshell, acyclicity requires that all possible cycles in $\mathit{graph}(p)$ are guarded. This ensures that cycles in $\mathit{graph}(p)$ do not lead to cycles in the ground graph. Additionally, acyclicity requires that programs are negation-guarded. This ensures that the relaxed grounding and the ground graph are well-defined. In the following, let p be a $(\Sigma, \mathbf{dom})$-ProbLog program.

Annotations. Annotations represent properties of the relations induced by the program p, and they are syntactically derived by analysing p's ground atoms and rules.

Let $a, a' \in \Sigma$ be two predicate symbols such that $|a| = |a'|$. A *disjointness annotation* $\mathit{DIS}(a, a')$ represents that the relations induced by a and a' (given p's relaxed grounding) are disjoint. We say that $\mathit{DIS}(a, a')$ can be derived from p iff no rules in p contain a or a' in their heads, and there is no $\overline{v} \in \mathbf{dom}^{|a|}$ where both $a(\overline{v})$ and $a'(\overline{v})$ appear as (possibly probabilistic) ground atoms in p. Hence, the relations induced by a and a' are disjoint.

Let $n \in \mathbb{N}$ and $A \subseteq \Sigma$ be a set of predicate symbols such that $|a| = 2n$ for all $a \in A$. An *ordering annotation* $\mathit{ORD}(A)$ represents that the transitive closure of the union of the relations induced by predicates in A given p's relaxed grounding is a strict partial order over $\mathbf{dom}^n$. The annotation $\mathit{ORD}(A)$ can be derived from the program p iff there is no rule $r \in p$ that contains any of the predicates in A in its head and the transitive closure of $\bigcup_{a \in A} \{((v_1, \ldots, v_n), (v_{n+1}, \ldots, v_{2n})) \mid \exists k.\ k{::}a(v_1, \ldots, v_{2n}) \in p\}$ is a strict partial order over $\mathbf{dom}^n$. Hence, the closure of the relation $\bigcup_{a \in A} \{((v_1, \ldots, v_n), (v_{n+1}, \ldots, v_{2n})) \mid a(v_1, \ldots, v_{2n}) \in \mathit{rg}(p)\}$ induced by the relaxed grounding is a strict partial order.

Let $a \in \Sigma$ be a predicate symbol and $K \subseteq \{1, \ldots, |a|\}$. A *uniqueness annotation* $\mathit{UNQ}(a, K)$ represents that the attributes in K are a primary key for the relation induced by a given the relaxed grounding. We say that $\mathit{UNQ}(a, K)$ can be derived from a program p iff no rule contains a in its head and for all $\overline{v}, \overline{v}' \in \mathbf{dom}^{|a|}$, if (1) $\overline{v}(i) = \overline{v}'(i)$ for all $i \in K$, and (2) there are k and k' such $k{::}a(\overline{v}) \in p$ and $k'{::}a(\overline{v}') \in p$, then $\overline{v} = \overline{v}'$. This ensures that whenever $a(\overline{v}), a(\overline{v}') \in \mathit{rg}(p)$ and $\overline{v}(i) = \overline{v}'(i)$ for all $i \in K$, then $\overline{v} = \overline{v}'$.

A Σ-*template* $\mathcal{T}$ is a set of annotations. In [36], we relax our syntactic rules for deriving annotations.

Example VI.3. We can derive $\mathit{DIS}(D, E)$ from the program in Example VI.1 since no rule generates facts for D and E and the relations defined by the ground atoms are $\{1\}$ and $\{2\}$. We can also derive $\mathit{ORD}(\{O\})$ since the relation defined by O's ground atoms is $\{(1, 2), (2, 3)\}$, whose transitive closure is a strict partial order. Finally, we can derive $\mathit{UNQ}(O, \{1\})$, $\mathit{UNQ}(O, \{2\})$, and $\mathit{UNQ}(O, \{1, 2\})$ since both arguments of O uniquely identify the tuples in the relation induced by O. ■

Unsafe structures. An unsafe structure models a part of the dependency graph that may introduce cycles in the ground graph. We define directed and undirected unsafe structures which may respectively introduce directed and undirected cycles in the ground graph.

A *directed unsafe structure* in $\mathit{graph}(p)$ is a directed cycle C in $\mathit{graph}(p)$. We say that a directed unsafe structure C *covers* a directed cycle C' iff C is equivalent to C'.

An *undirected unsafe structure* in $\mathit{graph}(p)$ is quadruple $\langle D_1, D_2, D_3, U \rangle$ such that (1) D_1, D_2, and D_3 are directed paths whereas U is an undirected path, (2) D_1 and D_2 start from the same node, (3) D_2 and D_3 end in the same node, and (4) $D_1 \cdot U \cdot D_3 \cdot D_2$ is an undirected cycle in $\mathit{graph}(p)$. We say that an unsafe structure $\langle D_1, D_2, D_3, U \rangle$ *covers* an undirected cycle U' in $\mathit{graph}(p)$ iff $D_1 \cdot U \cdot D_3 \cdot D_2$ is equivalent to U'.

Example VI.4. The cycle introduced by the rule r_c is captured by the directed unsafe structure $B \xrightarrow{r_c,1} B$, while the cycle introduced by r_a and r_b is captured by the structure $\langle A \xrightarrow{r_a,1} B, A \xrightarrow{r_b,1} B, \epsilon, \epsilon \rangle$, where ϵ denotes the empty path. ■

Connected Rules. A connected rule r ensures that a grounding of r is fully determined either by the assignment to the head's variables or to the variables of any literal in r's body. Formally, a strongly connected rule r guarantees that for any two groundings gr', gr'' of r, if $\mathit{head}(gr') = \mathit{head}(gr'')$, then $gr' = gr''$. In contrast, a weakly connected rule r guarantees that for any two groundings gr', gr'' of r, if $\mathit{body}(gr', i) = \mathit{body}(gr'', i)$ for some i, then $gr' = gr''$. This is done by exploiting uniqueness annotations and the rule's structure.

Before formalizing connected rules, we introduce join trees. A join tree represents how multiple predicate symbols in a rule share variables. In the following, let r be a rule and $\mathcal{T}$ be a template. A *join tree for a rule* r is a rooted labelled tree $(N, E, \mathit{root}, \lambda)$, where $N \subseteq \mathit{body}(r)$, E is a set of edges (i.e., unordered pairs over N^2), $\mathit{root} \in N$ is the tree's root, and λ is the labelling function. Moreover, we require that for all $n, n' \in N$, if $n \neq n'$ and $(n, n') \in E$, then $\lambda(n, n') = \mathit{vars}(n) \cap \mathit{vars}(n')$ and $\lambda(n, n') \neq \emptyset$. A join tree $(N, E, \mathit{root}, \lambda)$ *covers* a literal l iff $l \in N$. Given a join tree $J = (N, E, \mathit{root}, \lambda)$ and a node $n \in N$, the *support of* n, denoted $\mathit{support}(n)$, is the set $\mathit{vars}(\mathit{head}(r)) \cup \{x \mid (x = c) \in \mathit{cstr}(r) \wedge c \in \mathbf{dom}\} \cup \{\mathit{vars}(n') \mid n' \in \mathit{anc}(J, n)\}$, where $\mathit{anc}(J, n)$ is the set of n's

ancestors in J, i.e., the set of all nodes (different from n) on the path from *root* to n. A join tree $J = (N, E, root, \lambda)$ is $\mathcal{T}$*-strongly connected* iff for all positive literals $l \in N$, there is a set $K \subseteq \{i \mid \overline{x} = args(l) \wedge \overline{x}(i) \in support(l)\}$ such that $UNQ(pred(l), K) \in \mathcal{T}$ and for all negative literals $l \in N$, $vars(l) \subseteq support(l)$. In contrast, a join tree $(N, E, root, \lambda)$ is $\mathcal{T}$*-weakly connected* iff for all $(a(\overline{x}), a'(\overline{x}')) \in E$, there are $K \subseteq \{i \mid \overline{x}(i) \in L\}$ and $K' \subseteq \{i \mid \overline{x}'(i) \in L\}$ such that $UNQ(a, K), UNQ(a', K') \in \mathcal{T}$, where $L = \lambda(a(\overline{x}), a'(\overline{x}'))$.

We now formalize strongly and weakly connected rules. A rule r is $\mathcal{T}$*-strongly connected* iff there exist $\mathcal{T}$-strongly connected join trees $J_1, \ldots, J_n$ that cover all literals in r's body. This guarantees that for any two groundings gr', gr'' of r, if $head(gr') = head(gr'')$, then $gr' = gr''$.

Given a rule r, a set of literals L, and a template $\mathcal{T}$, a literal $l \in body(r)$ is $(r, \mathcal{T}, L)$*-strictly guarded* iff (1) $vars(l) \subseteq \bigcup_{l' \in L \cap body^+(r)} vars(l') \cup \{x \mid (x = c) \in cstr(r) \wedge c \in \mathbf{dom}\}$, and (2) there is a positive literal $a(\overline{x}) \in L$ and an annotation $UNQ(a, K) \in \mathcal{T}$ such that $\{\overline{x}(i) \mid i \in K\} \subseteq vars(l)$. A rule r is *weakly connected for* $\mathcal{T}$ iff there exists a $\mathcal{T}$-weakly connected join tree $J = (N, E, root, \lambda)$ such that $N \subseteq body^+(r)$, and all literals in $body(r) \setminus N$ are $(r, \mathcal{T}, N)$-strictly guarded. This guarantees that for any two groundings gr', gr'' of r, if $body(gr', i) = body(gr'', i)$ for some i, then $gr' = gr''$.

Example VI.5. Let $\mathcal{T}$ be the template from Example VI.3. The rule $r_c := B(y) \leftarrow B(x), \neg F(x), O(x, y)$ is $\mathcal{T}$-strongly connected. Indeed, the join tree having $O(x, y)$ as root and $B(x)$ and $\neg F(x)$ as leaves is such that (1) there is a uniqueness annotation $UNQ(O, \{2\})$ in $\mathcal{T}$ such that the second variable in $O(x, y)$ is included in those of r_c's head, (2) the variables in $B(x)$ and $\neg F(x)$ are a subset of those of their ancestors, and (3) the tree covers all literals in r_c's body. The rule is also $\mathcal{T}$-weakly connected: the join tree consisting only of $O(x, y)$ is $\mathcal{T}$-weakly connected and the literals $B(x)$ and $\neg F(x)$ are strictly guarded. Note that the rules r_a and r_b are trivially both strongly and weakly connected. ■

Guarded undirected structures. Guarded undirected structures ensure that undirected cycles in the dependency graph do not correspond to undirected cycles in the ground graph by exploiting disjointness annotations. Formally, an undirected unsafe structure $\langle D_1, D_2, D_3, U \rangle$ is *guarded by a template* $\mathcal{T}$ iff either (D_1, D_2) is $\mathcal{T}$-head-guarded or (D_2, D_3) is $\mathcal{T}$-tail-guarded.

A pair of non-empty paths (P_1, P_2) sharing the same initial node a is $\mathcal{T}$*-head guarded* iff (1) if $P_1 = P_2$, all rules in P_1 are weakly connected for $\mathcal{T}$, or (2) if $P_1 \neq P_2$, there is an annotation $DIS(pr, pr') \in \mathcal{T}$, a set $K \subseteq \{1, \ldots, |a|\}$, and a bijection $\nu : K \to \{1, \ldots, |pr|\}$ such that P_1 ν-downward links to pr and P_2 ν-downward links to pr'. Given two ground paths P_1' and P_2' corresponding to P_1 and P_2, the first condition ensures that $P_1' = P_2'$ whereas the second ensures that P_1' or P_2' are not in the ground graph.

Similarly, a pair of non-empty paths (P_1, P_2) sharing the same final node a is $\mathcal{T}$*-tail guarded* iff (1) if $P_1 = P_2$, all rules in P_1 are strongly connected for $\mathcal{T}$, or (2) if $P_1 \neq P_2$, there is an annotation $DIS(pr, pr') \in \mathcal{T}$, a set $K \subseteq \{1, \ldots, |a|\}$, and a bijection $\nu : K \to \{1, \ldots, |pr|\}$, such that P_1 ν-upward links to pr and P_2 ν-upward links to pr'.

Example VI.6. The only non-trivially guarded undirected cycle in the graph from Figure 6 is the one represented by the undirected unsafe structure $\langle A \xrightarrow{r_a,1} B, A \xrightarrow{r_b,1} B, \epsilon, \epsilon \rangle$. The structure is guarded since the paths $A \xrightarrow{r_a,1} B$ and $A \xrightarrow{r_b,1} B$ are head guarded by $DIS(D, E)$. Indeed, for the same ground atom $A(v)$, for some $v \in \{1, 2, 3\}$, only one of r_a and r_b can be applied since D and E are disjoint. ■

Guarded directed structures. Guarded directed structures exploit ordering annotations to ensure that directed cycles in the dependency graph do not correspond to directed cycles among ground atoms. A directed unsafe structure $pr_1 \xrightarrow{r_1,i_1} \ldots \xrightarrow{r_n,i_n} pr_1$ is *guarded by a template* $\mathcal{T}$ iff there is an annotation $ORD(O) \in \mathcal{T}$, integers $1 \leq y_1 < y_2 < \ldots < y_e = n$, literals $o_1(\overline{x}_1), \ldots, o_e(\overline{x}_e)$ (where $o_1, \ldots, o_e \in O$), a non-empty set $K \subseteq \{1, \ldots, |pr_1|\}$, and a bijection $\nu : K \to \{1, \ldots, {}^{|o|}/_2\}$ such that for each $0 \leq k < e$, (1) $pr_{y_k} \xrightarrow{r_{y_k}, i_{y_k}} \ldots \xrightarrow{r_{y_{k+1}-1}, i_{y_{k+1}-1}} pr_{y_{k+1}}$ ν-downward connects to $o_{k+1}(\overline{x}_{k+1})$, and (2) $pr_{y_{k+1}-1} \xrightarrow{r_{y_{k+1}-1}, i_{y_{k+1}-1}} pr_{y_{k+1}}$ ν'-upward connects to $o_{k+1}(\overline{x}_{k+1})$, where $\nu'(i) = \nu(x) + {}^{|o_1|}/_2$ for all $1 \leq i \leq {}^{|o_1|}/_2$, and $y_0 = 1$.

Example VI.7. The directed unsafe structure $B \xrightarrow{r_c,1} B$ is guarded by $ORD(\{O\})$ in the template from Example VI.3. Indeed, the strict partial order induced by O breaks the cycle among ground atoms belonging to B. In particular, the path $B \xrightarrow{r_c,1} B$ both downward links and upward links to $O(x, y)$; see Example VI.2. ■

Acyclic Programs. Let p be a negation-guarded program and $\mathcal{T}$ be the template containing all annotations that can be derived from p. We say that p is *acyclic* iff (a) for all undirected cycles U in $graph(p)$ that are not directed cycles, there is a $\mathcal{T}$-guarded undirected unsafe structure that covers U, and (b) for all directed cycles C in $graph(p)$, there is a $\mathcal{T}$-guarded directed unsafe structure that covers C. This ensures the absence of cycles in the ground graph.

Proposition VI.1. *Let p be a* ProbLog *program. If p is acyclic, then the ground graph of p is a forest of poly-trees.*

Example VI.8. The program p from Example VI.1 is acyclic. This is reflected in the ground graph in Figure 7. The program $q = p \cup \{E(1)\}$, however, is not acyclic: we cannot derive $DIS(D, E)$ from q and the undirected unsafe structure $\langle A \xrightarrow{r_a,1} B, A \xrightarrow{r_b,1} B, \epsilon, \epsilon \rangle$ is not guarded. As expected, q's ground graph contains an undirected cycle between $A(1)$ and $B(1)$, as shown in Figure 8. ■

Expressiveness. Acyclicity trades off the programs expressible in ProbLog for a tractable inference procedure. Acyclic programs, nevertheless, can encode many relevant probabilistic models.

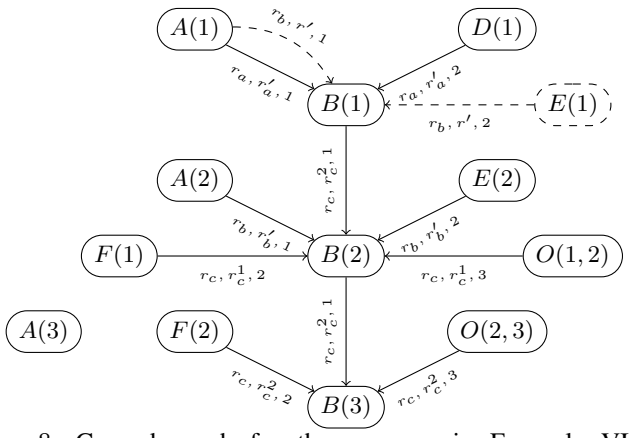
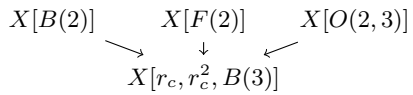


Fig. 8: Ground graph for the program in Example VI.1 extended with the atom $E(1)$. The additional edges and nodes are represented using dashed lines. The ground rules r'_a, r'_b, r^1_c, and r^2_c are as in Figure 7, and $r' = B(1) \leftarrow A(1), E(1)$.



Selected CPTs.

$X[B(2)]$	$X[F(2)]$	$X[O(2,3)]$	$X[r_c, r_c^2, B(3)]$ $\top$	$\bot$
$\top$	$\top$	$\top$	0	1
$\top$	$\top$	$\bot$	0	1
$\top$	$\bot$	$\top$	1	0
$\top$	$\bot$	$\bot$	0	1
$\bot$	$\top$	$\top$	0	1
$\bot$	$\top$	$\bot$	0	1
$\bot$	$\bot$	$\top$	0	1
$\bot$	$\bot$	$\bot$	0	1

Fig. 9: Portion of the resulting BN for the atoms $B(2)$, $F(2)$, and $O(2,3)$, the rule $r_c = B(y) \leftarrow B(x), \neg F(x), O(x,y)$, and the ground rule $r^2_c = B(3) \leftarrow B(2), \neg F(2), O(2,3)$, together with the CPT encoding r^2_c's semantics.

Proposition VI.2. *Any forest of poly-tree BNs can be represented as an acyclic* ProbLog *program.*

To clarify this proposition's scope, observe that poly-tree BNs are one of the few classes of BNs with tractable inference procedures. From Proposition VI.2, it follows that a large class of probabilistic models with tractable inference can be represented as acyclic programs. This supports our thesis that our syntactic constraints are not overly restrictive. In [36], we relax acyclicity to support a limited form of annotated disjunctions and rules sharing a part of their bodies, which are needed to encode the example from §V and for Proposition VI.2. We also provide all the proofs.

C. Inference Engine

Our inference algorithm for acyclic ProbLog programs consists of three steps: (1) we compute the relaxed grounding of p (cf. §VI-A), (2) we compile the relaxed grounding into a Bayesian Network (BN), and (3) we perform the inference using standard algorithms for poly-tree Bayesian Networks [48].

Encoding as BNs. We compile the relaxed grounding $rg(p)$ into the BN $bn(p)$. The boolean random variables in $bn(p)$ are as follows: (a) for each atom a in $rg(p)$ and ground literal a or $\neg a$ occurring in any $gr \in \bigcup_{r \in p} rg(p,r)$, there is a random variable $X[a]$, (b) for each rule $r \in p$ and each ground atom a such that there is $gr \in rg(p,r)$ satisfying $a = head(gr)$, there is a random variable $X[r,a]$, and (c) for each rule $r \in p$, each ground atom a, and each ground rule $gr \in rg(p,r)$ such that $a = head(gr)$, there is a random variable $X[r,gr,a]$.

The edges in $bn(p)$ are as follows: (a) for each ground atom a, rule r, and ground rule gr, there is an edge from $X[r,gr,a]$ to $X[r,a]$ and an edge from $X[r,a]$ to $X[a]$, and (b) for each ground atoms a and b, rule r, and ground rule gr, there is an edge from $X[b]$ to $X[r,gr,a]$ if b occurs in gr's body.

Finally, the Conditional Probability Tables (CPTs) of the variables in $bn(p)$ are as follows. The CPT of variables of the form $X[a]$ and $X[r,a]$ is just the OR of the values of their parents, i.e., the value is $\top$ with probability 1 iff at least one of the parents has value $\top$. For variables of the form $X[r, gr, a]$ such that $body(r) \neq \emptyset$, then the variable's CPT encode the semantics of the rule r, i.e., the value of $X[r,gr,a]$ is $\top$ with probability 1 iff all positive literals have value $\top$ and all negative literals have value $\bot$. In contrast, for variables of the form $X[r,gr,a]$ such that $body(r) = \emptyset$, then the variable has value $\top$ with probability v and $\bot$ with probability $1-v$, where r is of the form $v::a$ (if $r = a$ then $v = 1$).

To ensure that the size of the CPT of variables of the form $X[r,a]$ is independent of the size of the relaxed grounding, instead of directly connecting variables of the form $X[r,gr,a]$ with $X[r,a]$, we construct a binary tree of auxiliary variables where the leaves are all variables of the form $X[r,gr,a]$ and the root is the variable $X[r,a]$. Figure 9 depicts a portion of the BN for the program in Example VI.1.

Complexity. We now introduce the main result of this section.

Theorem VI.1. *The data complexity of inference for acyclic* ProbLog *programs is* Ptime.

This follows from (1) the relaxed grounding and the encoding can be computed in Ptime in terms of data complexity, (2) the encoding ensures that, for acyclic programs, the resulting Bayesian Network is a forest of poly-trees, and (3) inference algorithms for poly-tree BNs [48] run in polynomial time in the BN's size. In [36], we extend our encoding to handle additional features such as annotated disjunctions, and we prove its correctness and complexity.

VII. Angerona

Angerona is a DBIC mechanism that provably secures databases against probabilistic inferences. Angerona is parametrized by an AtkLog model representing the attacker's capabilities and it leverages ProbLog's inference capabilities.

A. Checking Query Security

Algorithm 1 presents Angerona. It takes as input a system state $s = \langle db, U, P \rangle$, a history h, the current query q issued by the user u, a system configuration C, and an AtkLog model

Algorithm 1: ANGERONA Enforcement Algorithm.

Input: A system state $s = \langle db, U, P\rangle$, a history h, an action $\langle u, q\rangle$, a system configuration C, and a C-ATKLOG model ATK.
Output: The security decision in $\{\top, \bot\}$.

```
begin
    for ⟨u, ψ, l⟩ ∈ secrets(P, u) do
        if secure(C, ATK, h, ⟨u, ψ, l⟩)
            if pox(C, ATK, h, ⟨u, q⟩)
                h' := h · ⟨⟨u, q⟩, ⊤, ⊤⟩
                if ¬secure(C, ATK, h', ⟨u, ψ, l⟩)
                    return ⊥
            if pox(C, ATK, h, ⟨u, ¬q⟩)
                h' := h · ⟨⟨u, q⟩, ⊤, ⊥⟩
                if ¬secure(C, ATK, h', ⟨u, ψ, l⟩)
                    return ⊥
    return ⊤

function secure(⟨D, Γ⟩, ATK, h, ⟨u, ψ, l⟩)
    p := ATK(u)
    for φ ∈ knowledge(h, u) do
        p := p ∪ PL(φ) ∪ {evidence(head(φ), true)}
    p := p ∪ PL(ψ)
    return [[p]]_D(head(ψ)) < l

function pox(⟨D, Γ⟩, ATK, h, ⟨u, ψ⟩)
    p := ATK(u)
    for φ ∈ knowledge(h, u) do
        p := p ∪ PL(φ) ∪ {evidence(head(φ), true)}
    p := p ∪ PL(ψ)
    return [[p]]_D(head(ψ)) > 0
```

ATK formalizing the users' beliefs. ANGERONA checks whether disclosing the result of the current query q may violate any secrets in $secrets(P, u)$. If this is the case, the algorithm concludes that q's execution would be insecure and returns $\bot$. Otherwise, it returns $\top$ and authorizes q's execution. Note that once we fix a configuration C and an ATKLOG model ATK, ANGERONA is a C-PDP as defined in §IV-C.

To check whether a query q may violate a secret $\langle u, \psi, l\rangle \in secrets(P, u)$, ANGERONA first checks whether the secret has been already violated. If this is not the case, ANGERONA checks whether disclosing q violates any secret. This requires checking that u's belief about the secret ψ stays below the threshold independently of the result of the query q; hence, we must ensure that u's belief is below the threshold both in case the query q holds in the actual database and in case q does not hold (this ensures that the access control decision itself does not leak information). ANGERONA, therefore, first checks whether there exists at least one possible database state where q is satisfied given h, using the procedure pox. If this is the case, the algorithm extends the current history h with the new event recording that the query q is authorized and its result is $\top$ and it checks whether u's belief about ψ is still below the corresponding threshold once q's result is disclosed, using the *secure* procedure. Afterwards, ANGERONA checks whether there exists at least a possible database state where q is not satisfied given h, it extends the current history h with another event representing that the query q does not hold, and it checks again whether disclosing that q does not hold in the current database state violates the secret. Note that checking whether there is a database state where q is (or is not) satisfied is essential to ensure that the conditioning that happens in the *secure* procedure is well-defined, i.e., the set of states we condition on has non-zero probability.

ANGERONA uses the *secure* subroutine to determine whether a secret's confidentiality is violated. This subroutine takes as input a system configuration, an ATKLOG model ATK, a history h, and a secret $\langle u, \psi, l\rangle$. It first computes the set $knowledge(h, u)$ containing all the authorized queries in the u-projection of h, i.e., $knowledge(h, u) = \{\phi \mid \exists i.\, h|_u(i) = \langle\langle u, \phi\rangle, \top, \top\rangle\} \cup \{\neg\phi \mid \exists i.\, h|_u(i) = \langle\langle u, \phi\rangle, \top, \bot\rangle\}$. Afterwards, it generates a PROBLOG program p by extending $ATK(u)$ with additional rules. In more detail, it translates each relational calculus sentence $\phi \in knowledge(h, u)$ to an equivalent set of PROBLOG rules $PL(\phi)$. The translation $PL(\phi)$ is standard [2]. For example, given a query $\phi = (A(1) \wedge B(2)) \vee \neg C(3)$, the translation $PL(\phi)$ consists of the rules $\{(h_1 \leftarrow A(1)), (h_2 \leftarrow B(2)), (h_3 \leftarrow \neg C(3)), (h_4 \leftarrow h_1, h_2), (h_5 \leftarrow h_4), (h_5 \leftarrow h_3)\}$, where $h_1, \ldots, h_5$ are fresh predicate symbols. We denote by $head(\phi)$ the unique predicate symbol associated with the sentence ϕ by the translation $PL(\phi)$. In our example, $head(\phi)$ is the fresh predicate symbol h_5. The algorithm then conditions the initial probability distribution $ATK(u)$ based on the sentences in $knowledge(h, u)$. This is done using *evidence statements*, which are special PROBLOG statements of the form $evidence(a, v)$, where a is a ground atom and v is either `true` or `false`; see Appendix A. For each sentence $\phi \in knowledge(h, u)$, the program p contains a statement $evidence(head(\phi), \texttt{true})$. Finally, the algorithm translates ψ to a set of logic programming rules and checks whether ψ's probability is below the threshold l.

The *pox* subroutine takes as input a system configuration, an ATKLOG model ATK, a history h, and a query $\langle u, \psi\rangle$. It determines whether there is a database db' that satisfies ψ and complies with the history $h|_u$. Internally, the routine again constructs a PROBLOG program starting from ATK, $knowledge(h, u)$, and ψ. Afterwards, it uses the program to check whether the probability of ψ given $h|_u$ is greater than 0.

Given a run $\langle s, h\rangle$ and a user u, the *secure* and *pox* subroutines condition u's initial beliefs based on the sentences in $knowledge(h, u)$, instead of using the set $[\![r]\!]_{\sim_u}$ as in the ATKLOG semantics. The key insight is that, as we prove in [36], the set of possible database states defined by the sentences in $knowledge(h, u)$ is equivalent to $[\![r]\!]_{\sim_u}$, which contains all database states derivable from the runs $r' \sim_u r$. This allows us to use PROBLOG to implement ATKLOG's semantics without explicitly computing $[\![r]\!]_{\sim_u}$.

Example VII.1. Let ATK be the attacker model in Example IV.4, u be the user *Mallory*, the database state be $s_{\{\texttt{A},\texttt{B},\texttt{C}\}}$, where `Alice`, `Bob`, and `Carl` have cancer, and the policy P be the one from Example IV.2. Furthermore, let $q_1, \ldots, q_4$ be the queries issued by *Mallory* in Example IV.3. ANGERONA permits the execution of the first two queries since they do

not violate the policy. In contrast, it denies the execution of the last two queries as they leak sensitive information. ■

Confidentiality. As we prove in [36], ANGERONA provides the desired security guarantees for any ATKLOG-attacker. Namely, it authorizes only those queries whose disclosure does not increase an attacker's beliefs in the secrets above the corresponding thresholds. ANGERONA also provides precise completeness guarantees: it authorizes all secrecy-preserving queries. Informally, a query $\langle u, q\rangle$ is *secrecy-preserving* given a run r and a secret $\langle u, \psi, l\rangle$ iff disclosing the result of $\langle u, q\rangle$ in any run $r' \sim_u r$ does not violate the secret.

Theorem VII.1. *Let a system configuration C and a C-*ATKLOG *model ATK be given, and let* ANGERONA *be the C-PDP f.* ANGERONA *provides data confidentiality with respect to C and $\lambda u \in \mathcal{U}.\ [\![ATK(u)]\!]_D$, and it authorizes all secrecy-preserving queries.*

Complexity. ANGERONA's complexity is dominated by the complexity of inference. We focus our analysis only on data complexity, i.e., the complexity when only the ground atoms in the PROBLOG programs are part of the input while everything else is fixed. A *literal query* is a query consisting either of a ground atom $a(\bar{c})$ or its negation $\neg a(\bar{c})$. We call an ATKLOG model *acyclic* if all belief programs in it are acyclic. Furthermore, a *literal secret* is a secret $\langle U, \phi, l\rangle$ such that ϕ is a literal query. We prove in [36] that for acyclic ATKLOG models, literal queries, and literal secrets, the PROBLOG programs produced by the $secure$ and pox subroutines are acyclic. We can therefore use our dedicated inference engine from §VI to reason about them. Hence, ANGERONA can be used to protect databases in PTIME in terms of data complexity. Literal queries are expressive enough to formulate queries about the database content such as "does $Alice$ have cancer?".

Theorem VII.2. *For all acyclic* ATKLOG *attackers, for all literal queries q, for all runs r whose histories contain only literal queries and contain only secrets expressed using literal queries,* ANGERONA*'s data complexity is* PTIME.

Discussion. Our tractability guarantees apply only to acyclic ATKLOG models, literal queries, and literal secrets. Nevertheless, ANGERONA can still handle relevant problems of interest. As stated in §VI, acyclic models are as expressive as poly-tree Bayesian Networks, one of the few classes of Bayesian Networks with tractable inference. Hence, for many probabilistic models that cannot be represented as acyclic ATKLOG models, exact probabilistic inference is intractable.

Literal queries are expressive enough to state simple facts about the database content. More complex (non-literal) queries can be simulated using (possibly large) sequences of literal queries. Similarly, policies with non-literal secrets can be implemented as sets of literal secrets, and the Boole–Fréchet inequalities [39] can be used to derive the desired thresholds. In both cases, however, our completeness guarantees hold only for the resulting literal queries, not for the original ones.

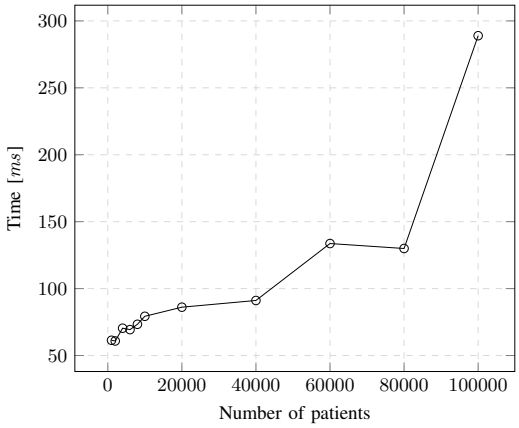


Fig. 10: ANGERONA execution time in seconds.

Finally, whenever our tractability constraints are violated, ANGERONA can still be used by directly using PROBLOG's inference capabilities. In this case, one would retain the security and completeness guarantees (Theorem VII.1) but lose the tractability guarantees (Theorem VII.2).

B. Implementation and Empirical Evaluation

To evaluate the feasibility of our approach in practice, we implemented a prototype of ANGERONA, available at [37]. The prototype implements our dedicated inference algorithm for acyclic PROBLOG programs (§VI), which computes the relaxed grounding of the input program p, constructs the Bayesian Networks BN, and performs the inference over BN using belief propagation [48]. For inference over BN, we rely on the GRRM library [66]. Observe that evidence statements in PROBLOG are encoded by fixing the values of the corresponding random variables in the BN. Note also that computing the relaxed grounding of p takes polynomial time in terms of data complexity, where the exponent is determined by p's rules. A key optimization is to pre-compute the relaxed grounding and construct BN off-line. This avoids grounding p and constructing the (same) Bayesian Network for each query. In our experiments we measure this time separately.

We use our prototype to study ANGERONA's efficiency and scalability. We run our experiments on a PC with an Intel i7 processor and 32GB of RAM. For our experiments, we consider the database schema from §IV. For the belief programs, we use the PROBLOG program given in §V, which can be encoded as an acyclic program when the parent-child relation is a poly-tree. We evaluate ANGERONA's efficiency and scalability in terms of the number of ground atoms in the belief programs. We generate synthetic belief programs containing 1,000 to 100,000 patients and for each of these instances, we generate 100 random queries of the form $R(\bar{t})$, where R is a predicate symbol and $\bar{t}$ is a tuple. For each instance and sequence of queries, we check the security of each query with our prototype, against a policy containing 100 randomly generated secrets specified as literal queries.

Figure 10 reports the execution times for our case study. Once the BN is generated, ANGERONA takes under 300

milliseconds, even for our larger examples, to check a query's security. During the initialization phase of our dedicated inference engine, we ground the original PROBLOG program and translate it into a BN. Most of the time is spent in the grounding process, whose data complexity is polynomial, where the polynomial's degree is determined by the number of free variables in the belief program. Our prototype uses a naive bottom-up grounding technique, and for our larger examples the initialization times is less than 2.5 minutes. We remark, however, that the initialization is performed just once per belief program. Furthermore, it can be done offline and its performance can be greatly improved by naive parallelization.

VIII. RELATED WORK

Database Inference Control. Existing DBIC approaches protect databases (either at design time [55], [56] or at runtime [14], [15], [40]) only against restricted classes of probabilistic dependencies, e.g., those arising from functional and multi-valued dependencies. ATKLOG, instead, supports arbitrary probabilistic dependencies, and even our acyclic fragment can express probabilistic dependencies that are not supported by [14], [15], [40], [55], [56]. Weise [71] proposes a DBIC framework, based on possibilistic logic, that formalizes secrets as sentences and expresses policies by associating bounds to secrets. Possibility theory differs from probability theory, which results in subtle differences. For instance, there is no widely accepted definition of conditioning for possibility distributions, cf. [11]. Thus, the probabilistic model from §II cannot be encoded in Weise's framework [71].

Statistical databases store information associated to different individuals and support queries that return statistical information about an entire population [16]. DBIC solutions for statistical databases [3], [16], [22]–[24] prevent leakages of information about single individuals while allowing the execution of statistical queries. These approaches rely on various techniques, such as perturbating the original data, synthetically generating data, or restricting the data on which the queries are executed. Instead, we protect specific secrets in the presence of probabilistic data dependencies and we return the original query result, without modifications, if it is secure.

Differential Privacy. Differential Privacy [25], [26] is widely used for privacy-preserving data analysis. Systems such as ProPer [27] or PINQ [54] provide users with automated ways to perform differentially private computations. A differentially private computation guarantees that the presence (or absence) of an individual's data in the input data set affects the probability of the computation's result only in limited way, i.e., by at most a factor e^{ϵ} where ϵ is a parameter controlling the privacy-utility trade-off. While differential privacy does not make any assumption about the attacker's beliefs, we assume that the attacker's belief is known and we guarantee that for all secrets in the policy, no user can increase his beliefs, as specified in the attacker model, over the corresponding thresholds by interacting with the system.

Information Flow Control. Quantified Information Flow [6], [17], [49], [51] aims at quantifying the amount of information leaked by a program. Instead of measuring the amount of leaked information, we focus on restricting the information that an attacker may obtain about a set of given secrets.

Non-interference has been extended to consider probabilities [5], [58], [70] for concurrent programs. Our security notion, instead, allows those leaks that do not increase an attacker's beliefs in a secret above the threshold, and it can be seen as a probabilistic extension of *opacity* [60], which allows any leak except leaking whether the secret holds.

Mardziel et al. [52] present a general DBIC architecture, where users' beliefs are expressed as probabilistic programs, security requirements as threshold on these beliefs, and the beliefs are updated in response to the system's behaviour. Our work directly builds on top of this architecture. However, instead of using an imperative probabilistic language, we formalize beliefs using probabilistic logic programming, which provides a natural and expressive language for formalizing dependencies arising in the database setting, e.g., functional and multi-valued dependencies, as well as common probabilistic models, like Bayesian Networks.

Mardziel et al. [52] also propose a DBIC mechanism based on abstract interpretation. They do not provide any precise complexity bound for their mechanism. Their algorithm's complexity class, however, appears to be intractable, since they use a probabilistic extension of the polyhedra abstract domain, whose asymptotic complexity is exponential in the number of program variables [62]. In contrast, ANGERONA exploits our inference engine for acyclic programs to secure databases against a practically relevant class of probabilistic inferences, and it provides precise tractability and completeness guarantees.

We now compare (unrestricted) ATKLOG with the imperative probabilistic language used in [52]. ATKLOG allows one to concisely encode probabilistic properties specifying relations between tuples in the database. For instance, a property like "the probability of $A(x)$ is $1/2^n$, where n is the number of tuples (x, y) in B" can be encoded as $1/2::A(x) \leftarrow B(x, y)$. Encoding this property as an imperative program is more complex; it requires a **for** statement to iterate over all variables representing tuples in B and an **if** statement to filter the tuples. In contrast to [52], ATKLOG provides limited support for numerical constraints (as we support only finite domains). Mardziel et al. [52] formalize queries as imperative probabilistic programs. They can, therefore, also model probabilistic queries or the use of randomization to limit disclosure. While all these features are supported by ATKLOG, our goal is to protect databases from attackers that use standard query languages like SQL. Hence, we formalize queries using relational calculus and ignore probabilistic queries. Similarly to [52], our approach can be extended to handle some uncertainty on the attackers' capabilities. In particular, we can associate to each user a finite number of possible beliefs, instead of a single one. However, like [52], we cannot handle infinitely many alternative beliefs.

Probabilistic Programming. Probabilistic programming is an active area of research [35]. Here, we position PROBLOG with

respect to expressiveness and inference. Similarly to [33], [52], PROBLOG can express only discrete probability distributions, and it is less expressive than languages supporting continuous distributions [32], [34], [61]. Current exact inference algorithms for probabilistic programs are based on program analysis techniques, such as symbolic execution [32], [61] or abstract interpretation [52]. In this respect, we present syntactic criteria that *ensure* tractable inference for PROBLOG. Sampson et al. [59] symbolically execute probabilistic programs and translate them to BNs to verify probabilistic assertions. In contrast, we translate PROBLOG programs to BNs to perform exact inference and our translation is tailored to work together with our acyclicity constraints to allow tractable inference.

IX. CONCLUSION

Effectively securing databases that store data with probabilistic dependencies requires an expressive language to capture the dependencies and a tractable enforcement mechanism. To address these requirements, we developed ATKLOG, a formal language providing an expressive and concise way to represent attackers' beliefs while interacting with the system. We leveraged this to design ANGERONA, a provably secure DBIC mechanism that prevents the leakage of sensitive information in the presence of probabilistic dependencies. ANGERONA is based on a dedicated inference engine for a fragment of PROBLOG where exact inference is tractable.

We see these results as providing a foundation for building practical protection mechanisms, which include probabilistic dependencies, as part of real-world database systems. As future work, we plan to extend our framework to dynamic settings where the database and the policy change. We also intend to explore different fragments of PROBLOG and relational calculus for which exact inference is practical.

Acknowledgments. We thank Ognjen Maric, Dmitriy Traytel, Der-Yeuan Yu, and the anonymous reviewers for their comments.

REFERENCES

[1] "ProbLog – Probabilistic Programming," Online at http://dtai.cs.kuleuven.be/problog/index.html.

[2] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of databases*. Addison-Wesley Reading, 1995, vol. 8.

[3] N. R. Adam and J. C. Worthmann, "Security-control methods for statistical databases: a comparative study," *ACM Computing Surveys (CSUR)*, vol. 21, no. 4, pp. 515–556, 1989.

[4] K. Ahmed, A. A. Emran, T. Jesmin, R. F. Mukti, M. Z. Rahman, and F. Ahmed, "Early detection of lung cancer risk using data mining," *Asian Pacific Journal of Cancer Prevention*, vol. 14, no. 1, pp. 595–598, 2013.

[5] A. Aldini, "Probabilistic information flow in a process algebra," in *Proceedings of the 12th International Conference on Concurrency Theory*. Springer, 2001, pp. 152–168.

[6] M. Alvim, M. Andrés, and C. Palamidessi, "Probabilistic information flow," in *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 2010, pp. 314–321.

[7] A. Askarov and S. Chong, "Learning is change in knowledge: Knowledge-based security for dynamic policies," in *Proceedings of the 25th IEEE Computer Security Foundations Symposium*. IEEE, 2012, pp. 308–322.

[8] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *Proceedings of the 28th IEEE Symposium on Security and Privacy*. IEEE, 2007, pp. 207–221.

[9] V. Bárány, B. Ten Cate, and M. Otto, "Queries with guarded negation," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1328–1339, 2012.

[10] P. A. Bonatti, S. Kraus, and V. Subrahmanian, "Foundations of secure deductive databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 7, no. 3, pp. 406–422, 1995.

[11] B. Bouchon-Meunier, G. Coletti, and C. Marsala, "Independence and possibilistic conditioning," *Annals of Mathematics and Artificial Intelligence*, vol. 35, no. 1, pp. 107–123, 2002.

[12] A. Brodsky, C. Farkas, and S. Jajodia, "Secure databases: Constraints, inference channels, and monitoring disclosures," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 6, pp. 900–919, 2000.

[13] Centers for Medicare & Medicaid Services, "The Health Insurance Portability and Accountability Act of 1996 (HIPAA)," Online at http://www.cms.hhs.gov/hipaa/, 1996.

[14] Y. Chen and W. W. Chu, "Database security protection via inference detection," in *International Conference on Intelligence and Security Informatics*. Springer, 2006, pp. 452–458.

[15] ——, "Protection of database security via collaborative inference detection," *IEEE Transactions on Knowledge and Data Engineering*, vol. 20, no. 8, pp. 1013–1027, 2008.

[16] F. Y. Chin and G. Ozsoyoglu, "Auditing and inference control in statistical databases," *IEEE Transactions on Software Engineering*, vol. 8, no. 6, pp. 574–582, 1982.

[17] D. Clark, S. Hunt, and P. Malacaria, "A static analysis for quantifying information flow in a simple imperative language," *Journal of Computer Security*, vol. 15, no. 3, pp. 321–371, 2007.

[18] M. R. Clarkson, A. C. Myers, and F. B. Schneider, "Belief in information flow," in *Proceedings of the 18th IEEE Workshop on Computer Security Foundations*. IEEE, 2005, pp. 31–45.

[19] ——, "Quantifying information flow with beliefs," *Journal of Computer Security*, vol. 17, no. 5, pp. 655–701, 2009.

[20] L. De Raedt and A. Kimmig, "Probabilistic (logic) programming concepts," *Machine Learning*, vol. 100, no. 1, pp. 5–47, 2015.

[21] L. De Raedt, A. Kimmig, and H. Toivonen, "Problog: A probabilistic prolog and its application in link discovery," in *Proceedings of the 20th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 2007, pp. 2468–2473.

[22] D. E. Denning, "Secure statistical databases with random sample queries," *ACM Transactions on Database Systems (TODS)*, vol. 5, no. 3, pp. 291–315, 1980.

[23] D. Dobkin, A. K. Jones, and R. J. Lipton, "Secure databases: protection against user influence," *ACM Transactions on Database Systems*, vol. 4, no. 1, pp. 97–106, 1979.

[24] J. Domingo-Ferrer, *Inference control in statistical databases: From theory to practice*. Springer, 2002, vol. 2316.

[25] C. Dwork, "Differential privacy," in *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming*. Springer, 2006, pp. 1–12.

[26] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," *Foundations and Trends in Theoretical Computer Science*, vol. 9, no. 3-4, pp. 211–407, 2014.

[27] H. Ebadi, D. Sands, and G. Schneider, "Differential privacy: Now it's getting personal," in *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2015, pp. 69–81.

[28] European Parliament, "General Data Protection Regulation (2016/679)," Online at http://eur-lex.europa.eu/eli/reg/2016/679/oj, 2016.

[29] A. Evfimievski, R. Fagin, and D. Woodruff, "Epistemic privacy," *Journal of ACM*, vol. 58, no. 1, pp. 2:1–2:45, 2010.

[30] C. Farkas and S. Jajodia, "The inference problem: a survey," *ACM SIGKDD Explorations Newsletter*, vol. 4, no. 2, pp. 6–11, 2002.

[31] D. Fierens, G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. De Raedt, "Inference and learning in probabilistic logic programs using weighted boolean formulas," *Theory and Practice of Logic Programming*, vol. 15, no. 03, pp. 358–401, 2015.

[32] T. Gehr, S. Misailovic, and M. Vechev, "Psi: Exact symbolic inference for probabilistic programs," in *Proceedings of the 28th International Conference on Computer Aided Verification*. Springer, 2016, pp. 62–83.

[33] L. Getoor, *Introduction to statistical relational learning*. MIT press, 2007.

[34] A. D. Gordon, T. Graepel, N. Rolland, C. Russo, J. Borgstrom, and J. Guiver, "Tabular: A schema-driven probabilistic programming language," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2014, pp. 321–334.

[35] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani, "Probabilistic programming," in *Proceedings of the Conference on The Future of Software Engineering*. ACM, 2014, pp. 167–181.

[36] M. Guarnieri, S. Marinovic, and D. Basin, "Securing databases from probabilistic inference — extended version," Online at https://arxiv.org/abs/1706.02473.

[37] ——, "Securing databases from probabilistic inference — prototype," Online at http://www.infsec.ethz.ch/research/projects/FDAC.html.

[38] ——, "Strong and provably secure database access control," in *Proceedings of the 1st European Symposium on Security and Privacy*. ACM, 2016, pp. 163–178.

[39] T. Hailperin, "Probability logic," *Notre Dame Journal of Formal Logic*, vol. 25, no. 3, pp. 198–212, 1984.

[40] J. Hale and S. Shenoi, "Catalytic inference analysis: Detecting inference threats due to knowledge discovery," in *Proceedings of the 18th IEEE Symposium on Security and Privacy*. IEEE, 1997, pp. 188–199.

[41] J. He, W. W. Chu, and Z. V. Liu, "Inferring privacy information from social networks," in *International Conference on Intelligence and Security Informatics*. Springer, 2006, pp. 154–165.

[42] D. Hedin and A. Sabelfeld, "A perspective on information-flow control," in *Proceedings of the 2011 Marktoberdorf Summer School*.

[43] T. H. Hinke, H. S. Delugach, and R. P. Wolf, "Protecting databases from inference attacks," *Computers & Security*, vol. 16, no. 8, pp. 687–708, 1997.

[44] M. Humbert, E. Ayday, J.-P. Hubaux, and A. Telenti, "Addressing the concerns of the lacks family: Quantification of kin genomic privacy," in *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2013, pp. 1141–1152.

[45] G. Kabra, R. Ramamurthy, and S. Sudarshan, "Redundancy and information leakage in fine-grained access control," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. ACM, 2006.

[46] V. Katos, D. Vrakas, and P. Katsaros, "A framework for access control with inference constraints," in *Proceedings of 35th IEEE Annual Conference on Computer Software and Applications*. IEEE, 2011, pp. 289–297.

[47] K. Kenthapadi, N. Mishra, and K. Nissim, "Simulatable auditing," in *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM, 2005, pp. 118–127.

[48] D. Koller and N. Friedman, *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[49] B. Köpf and D. Basin, "An information-theoretic model for adaptive side-channel attacks," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*. ACM, 2007, pp. 286–296.

[50] S. L. Lauritzen and N. A. Sheehan, "Graphical models for genetic analyses," *Statistical Science*, vol. 18, no. 4, pp. 489–514, 2003.

[51] G. Lowe, "Quantifying information flow," in *Proceedings of the 15th IEEE Workshop on Computer Security Foundations*. IEEE, 2002, pp. 18–31.

[52] P. Mardziel, S. Magill, M. Hicks, and M. Srivatsa, "Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation," *Journal of Computer Security*, vol. 21, no. 4, pp. 463–532, 2013.

[53] W. Mathew, R. Raposo, and B. Martins, "Predicting future locations with hidden markov models," in *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*. ACM, 2012, pp. 911–918.

[54] F. D. McSherry, "Privacy integrated queries: an extensible platform for privacy-preserving data analysis," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 2009, pp. 19–30.

[55] M. Morgenstern, "Security and inference in multilevel database and knowledge-base systems," in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of data*. ACM, 1987, pp. 357–373.

[56] ——, "Controlling logical inference in multilevel database systems," in *Proceedings of the 9th IEEE Symposium on Security and Privacy*. IEEE, 1988, pp. 245–255.

[57] X. Qian, M. E. Stickel, P. D. Karp, T. F. Lunt, and T. D. Garvey, "Detection and elimination of inference channels in multilevel relational database systems," in *Proceedings of the 14th IEEE Symposium on Security and Privacy*. IEEE, 1993, pp. 196–205.

[58] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multi-threaded programs," in *Proceedings of the 13th IEEE Workshop on Computer Security Foundations*. IEEE, 2000, pp. 200–214.

[59] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze, "Expressing and verifying probabilistic assertions," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2014, pp. 112–122.

[60] D. Schoepe and A. Sabelfeld, "Understanding and enforcing opacity," in *Proceedings of the 28th IEEE Computer Security Foundations Symposium*. IEEE, 2015, pp. 539–553.

[61] C.-c. Shan and N. Ramsey, "Exact bayesian inference by symbolic disintegration," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 2017, pp. 130–144.

[62] G. Singh, M. Püschel, and M. Vechev, "Fast polyhedra abstract domain," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 2017, pp. 46–59.

[63] T.-A. Su and G. Ozsoyoglu, "Controlling FD and MVD inferences in multilevel relational database systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 3, no. 4, pp. 474–485, 1991.

[64] ——, "Data dependencies and inference control in multilevel relational database systems," in *Proceedings of the 7th IEEE Symposium on Security and Privacy*. IEEE, 1987, pp. 202–211.

[65] D. Suciu, D. Olteanu, C. Ré, and C. Koch, "Probabilistic databases," *Synthesis Lectures on Data Management*, vol. 3, no. 2, pp. 1–180, 2011.

[66] C. Sutton, "GRMM: GRaphical Models in Mallet," Online at http://mallet.cs.umass.edu/grmm/.

[67] M. Thuraisingham, "Security checking in relational database management systems augmented with inference engines," *Computers & Security*, vol. 6, no. 6, pp. 479–492, 1987.

[68] T. S. Toland, C. Farkas, and C. M. Eastman, "The inference problem: Maintaining maximal availability in the presence of database updates," *Computers & Security*, vol. 29, no. 1, pp. 88–103, 2010.

[69] P. J. Villeneuve and Y. Mao, "Lifetime probability of developing lung cancer, by smoking status, Canada," *Cancer Journal of Public Health*, vol. 85, no. 6, pp. 385–388, 1994.

[70] D. Volpano and G. Smith, "Probabilistic noninterference in a concurrent language," *Journal of Computer Security*, vol. 7, no. 2-3, pp. 231–253, 1999.

[71] L. Wiese, "Keeping secrets in possibilistic knowledge bases with necessity-valued privacy policies," in *Computational Intelligence for Knowledge-Based Systems Design*. Springer, 2010, pp. 655–664.

[72] R. W. Yip and E. Levitt, "Data level inference detection in database systems," in *Proceedings of the 11th IEEE Computer Security Foundations Workshop*. IEEE, 1998, pp. 179–189.

APPENDIX A
PROBLOG

Here, we provide a formal account of PROBLOG, which follows [20], [21], [31]. In addition to PROBLOG's semantics, we present here again some (revised) aspects of PROBLOG's syntax, which we introduced in §V. As mentioned in §V, we restrict ourselves to the function-free fragment of PROBLOG.

Syntax. As introduced in §V, a $(\Sigma, \mathbf{dom})$-*probabilistic atom* is an atom $a \in \mathcal{A}_{\Sigma,\mathbf{dom}}$ annotated with a value $0 \leq v \leq 1$, denoted $v{::}a$. If $v = 1$, then we write $a(\overline{c})$ instead of $1{::}a(\overline{c})$. A $(\Sigma, \mathbf{dom})$-PROBLOG *program* is a finite set of ground probabilistic $(\Sigma, \mathbf{dom})$-atoms and $(\Sigma, \mathbf{dom})$-rules. Note that ground atoms $a \in \mathcal{A}_{\Sigma,\mathbf{dom}}$ are represented as $1{::}a$. Observe also that rules do not involve probabilistic atoms (as formalized in §V). This is without loss of generality: as we show below probabilistic rules and annotated disjunctions can be represented using only probabilistic atoms and non-probabilistic rules. We denote by $prob(p)$ the set of all probabilistic ground atoms $v{::}a$ in p, i.e., $prob(p) := \{v{::}a \in$

$p \mid 0 \leq v \leq 1 \wedge a \in \mathcal{A}_{\Sigma,\mathbf{dom}}\}$, and by $rules(p)$ the non-probabilistic rules in p, i.e., $rules(p) := p \backslash prob(p)$. As already stated in §V, we consider only programs p that do not contain negative cycles in the rules. Finally, we say that a PROBLOG program p is a *logic program* iff $v = 1$ for all $v::a \in prob(p)$, i.e., p does not contain probabilistic atoms.

Semantics. Given a $(\Sigma, \mathbf{dom})$-PROBLOG program p, a *p-grounded instance* is a PROBLOG program $A \cup R$, where the set of ground atoms A is a subset of $\{a \mid v::a \in prob(p)\}$ and $R = rules(p)$. Informally, a grounded instance of p is one of the logic programs that can be obtained by selecting some of the probabilistic atoms in p and keeping all rules in p. A *p-probabilistic assignment* is a total function associating to each probabilistic atom $v::a$ in $prob(p)$ a value in $\{\top, \bot\}$. We denote by $A(p)$ the set of all p-probabilistic assignments. The *probability* of a p-probabilistic assignment f is $prob(f) = \Pi_{v::a \in prob(p)} \left(\Pi_{f(v::a)=\top} v \cdot \Pi_{f(v::a)=\bot}(1-v)\right)$. Given a p-probabilistic assignment f, $instance(p, f)$ denotes the p-grounded instance $\{a \mid \exists v.\, f(v::a) = \top\} \cup rules(p)$. Finally, given a p-grounded instance p', $WFM(p')$ denotes the well-founded model associated with the logic program p', as defined by standard logic programming semantics [2].

The semantics of a $(\Sigma, \mathbf{dom})$-PROBLOG program p is defined as a probability distribution over all possible p-grounded instances. Note that PROBLOG's semantics relies on the *closed world assumption*, namely every fact that is not in a given model is considered false. The semantics of p, denoted by $[\![p]\!]$, is as follows: $[\![p]\!](p') = \sum_{f \in F_{p,p'}} prob(f)$, where $F_{p,p'} = \{f \in A(p) \mid p' = instance(p, f)\}$. We remark that a $(\Sigma, \mathbf{dom})$-PROBLOG program p implicitly defines a probability distribution over $(\Sigma, \mathbf{dom})$-structures. Indeed, the probability of a given $(\Sigma, \mathbf{dom})$-structure s is the sum of the probabilities of all p-grounded instances whose well-founded model is s. With a slight abuse of notation, we extend the semantics of p to $(\Sigma, \mathbf{dom})$-structures and ground atoms as follows: $[\![p]\!](s) = \sum_{f \in \mathcal{M}(p,s)} prob(f)$, where s is a $(\Sigma, \mathbf{dom})$-structure and $\mathcal{M}(p, s)$ is the set of all assignments f such that $WFM(instance(p, f)) = s$. Finally, p's semantics can be lifted to sentences as follows: $[\![p]\!](\phi) = \Sigma_{s \in [\![\phi]\!]} [\![p]\!](s)$, where $[\![\phi]\!] = \{s \in \Omega_D^\Gamma \mid [\phi]^s = \top\}$.

Evidence. PROBLOG supports expressing *evidence* inside programs [20]. To express evidence, i.e., to condition a distribution on some event, we use statements of the form $evidence(a, v)$, where a is a ground atom and $v \in \{\texttt{true}, \texttt{false}\}$. Let p be a $(\Sigma, \mathbf{dom})$-PROBLOG program p with evidence $evidence(a_1, v_1), \ldots, evidence(a_n, v_n)$, and p' be the program without the evidence statements. Furthermore, let $POX(p')$ be the set of all $(\Sigma, \mathbf{dom})$-structures s complying with the evidence, i.e., the set of all states s such that a_i holds in s iff $v_i = \texttt{true}$. Then, $[\![p]\!](s)$, for a $(\Sigma, \mathbf{dom})$-structure $s \in POX(p)$, is $[\![p']\!](s) \cdot \left(\sum_{s' \in POX(p)} [\![p']\!](s')\right)^{-1}$.

Syntactic Sugar. Following [20], [21], [31], we extend PROBLOG programs with two additional constructs: annotated disjunctions and probabilistic rules. As shown in [20], these constructs are just syntactic sugar. A *probabilistic rule* is a PROBLOG rule where the head is a probabilistic atom. The probabilistic rule $v::h \leftarrow l_1, \ldots, l_n$ can be encoded using the additional probabilistic atoms $v::sw(\_)$ and the rule $h \leftarrow l_1, \ldots, l_n, sw(\overline{x})$, where sw is a fresh predicate symbol, $\overline{x}$ is the tuple containing the variables in $vars(h) \cup \bigcup_{1 \leq i \leq n} vars(l_i)$, and $v::sw(\_)$ is a shorthand representing the fact that there is a probabilistic atom $v::sw(\overline{t})$ for each tuple $\overline{t} \in \mathbf{dom}^{|\overline{x}|}$.

An *annotated disjunction* $v_1::a_1; \ldots; v_n::a_n$, where $a_1, \ldots, a_n$ are ground atoms and $\left(\sum_{1 \leq i \leq n} v_i\right) \leq 1$, denotes that $a_1, \ldots, a_n$ are mutually exclusive probabilistic events happening with probabilities $v_1, \ldots, v_n$. It can be encoded as:

$$
\begin{aligned}
&p_1::sw_1(\_)\\
&\quad\vdots\\
&p_n::sw_n(\_)\\
&a_1(\overline{t}_1) \leftarrow sw_1(\overline{t}_1)\\
&\quad\vdots\\
&a_n(\overline{t}_n) \leftarrow \neg sw_1(\overline{t}_1), \ldots, \neg sw_{n-1}(\overline{t}_{n-1}), sw_n(\overline{t}_n),
\end{aligned}
$$

where each p_i, for $1 \leq i \leq n$, is $v_i \cdot \left(1 - \sum_{1 \leq j < i} v_j\right)^{-1}$. Probabilistic rules can be easily extended to support annotated disjunctions in their heads.

Example A.1. Let Σ be a first-order signature with two predicate symbols V and W, both with arity 1, $\mathbf{dom}$ be the domain $\{a, b\}$, and p be the program consisting of the facts $1/4::T(a)$ and $1/2::T(b)$, the annotate disjunction $1/4::W(a); 1/2::W(b)$, and the rule $1/2::T(x) \leftarrow W(x)$.

The probability associated to each $(\Sigma, \mathbf{dom})$-structure by the program p is shown in the following table.

		W			
		$\emptyset$	$\{a\}$	$\{b\}$	$\{a, b\}$
T	$\emptyset$	$3/32$	$3/64$	$3/32$	0
	$\{a\}$	$1/32$	$5/64$	$1/32$	0
	$\{b\}$	$3/32$	$3/64$	$9/32$	0
	$\{a, b\}$	$1/32$	$5/64$	$3/32$	0

The empty structure has probability $3/32$. The only grounded instance whose well-formed model is the empty database is the instance i_1 that does not contain grounded atoms. Its probability is $3/32$ because the probability that $T(a)$ is not in i_1 is $3/4$, the probability that $T(b)$ is not in i_1 is $1/2$, and the probability that neither $W(a)$ nor $W(b)$ are in i_1 is $1/4$ and all these events are independent.

The probability of some structures is determined by more than one grounded instance. For example, the probability of the structure s where $s(T) = \{a, b\}$ and $s(W) = \{a\}$ is $5/64$. There are two grounded instances i_2 and i_3 whose well-founded model is s. The instance i_2 has probability $1/16$ and it consists of the atoms $T(b), W(a), sw(a)$ and the rule $T(x) \leftarrow W(x), sw(x)$, whereas the instance i_3 has probability $1/64$ and it consists of the atoms $T(a), T(b), W(a)$ and the rule $T(x) \leftarrow W(x), sw(x)$. Note that before computing the ground instances, we translated probabilistic rules and annotated disjunctions into standard PROBLOG rules. ■