Decentralized Composite Access Control

Petar Tsankov, Srdjan Marinovic, Mohammad Torabi Dashti, and David Basin

Institute of Information Security, ETH Zurich, Switzerland
{ptsankov,srdanm,torabidm,basin}@inf.ethz.ch

Abstract. Formal foundations for access control policies with both authority delegation and policy composition operators are partial and limited. Correctness guarantees cannot therefore be formally stated and verified for decentralized composite access control systems, such as those based on XACML 3. To address this problem we develop a formal policy language BelLog that can express both delegation and composition operators. We illustrate, through examples, how BelLog can be used to specify practical policies. Moreover, we present an analysis framework for reasoning about BelLog policies and we give decidability and complexity results for policy entailment and policy containment in BelLog.

1 Introduction

We present the first formal language for specifying and reasoning about *decentralized composite* access control policies, which are policies that require both authority delegation and policy compositions. Below, we illustrate these concepts, and motivate the need for their formal study.

Consider a simple grid system. The grid owner allows *privileged* clients to issue access control policies for the grid's storage space by delegating the authority over the storage resources to them. Privileged clients issue policies, and may also further delegate this authority. To decide who can access storage resources, the grid owner composes the collected policies using different composition operators, such as permit-override (permit if at least one client grants access), majority voting (permit if most clients grant access), etc. This example demonstrates how modern access control systems require both authority delegation and policy composition features, hence going beyond composition-only systems, e.g. those based on XACML 2, and delegation-only systems, such as KeyNote 2 [1]. Real-world examples include grid resource sharing systems [2], electronic health record management [3] and highly distributed Web services [4]. To cater for such decentralized composite access control systems, the industry has recently released the XACML 3 standard.

The need for a formal foundation is evident: Without it, one cannot precisely define how existing and future decentralized composite access control systems should behave (e.g. the ones built upon XACML 3 implementations). Furthermore, formal guarantees about the correctness of decentralized composite policies, e.g. by answering policy entailment and containment questions, cannot be derived. The existing formal access control languages fall short in this regard.

M. Abadi and S. Kremer (Eds.): POST 2014, LNCS 8414, pp. 245–264, 2014.

They either express authority delegation or policy composition, but not both together; see the related work.

Contributions. We are the first to address the problem of formally specifying and reasoning about decentralized composite policies. We develop a novel logic programming language, dubbed BelLog, for constructing decentralized composite policy languages. BelLog is an extension of Datalog [5], where the truth values come from Belnap's four-valued logic [6]. All delegation languages based on Datalog can therefore be mapped to BelLog. Furthermore, BelLog is more expressive than the existing multi-valued policy algebras, such as PBel [7] and PTaCL [8].

Through examples, we illustrate how decentralized composite policies can be encoded in BelLog. We also present syntactic extensions of BelLog that ease the specification of common policy composition and authority delegation idioms, for instance: permit-override, only-one-applicable, agreement, hand-off trust application, transitive delegation, etc.

We present a policy analysis framework for verifying policies written in BelLog, and demonstrate how different policy analysis questions are used to reason about a policy's behavior in some or all system configurations. We show that verifying BelLog policies for a given system configuration is in PTIME, and verification for all possible system configurations of a finite domain of subjects and objects is in co-NP-complete. We furthermore identify a useful fragment of BelLog where verification for all possible system configurations for infinitely many subjects and objects belongs to co-NEXP.

Finally, BelLog can be used as a four-valued logic programming language for reasoning with inconsistent and incomplete knowledge. BelLog and its decision procedures are therefore of independent interest.

Related Work. The closest related works to BelLog are policy algebras, formal delegation languages, and XACML 3, which is an informal policy language.

Policy algebras—such as PBel [7], PTaCL [8], and D-Algebra [9]—are languages for composing a set of policies. A composite policy is a tree, where the internal nodes are composition operators, and the leaf nodes are core policies. Existing policy algebras cannot express arbitrarily long delegation chains and therefore cannot be used for decentralized composite access control. Moreover, they lack operators for composing *intensionally* defined policy sets, i.e. policy sets that are not fixed at the policy specification time; see §4.

Delegation languages—such as KeyNote2 [1], DKAL [10], SecPAL [11], RT [12], GP [13], and DCC [14]—allow a policy writer to delegate to other principals authority over attributes and policy decisions. In contrast to BelLog, these languages support only the permit-override operator for composing policies. Although the permit-override operator is sufficient in their access control setup, this is not the case for decentralized composite policies. Most existing delegation languages are founded on logic programming. We remark that although many-valued extensions for logic programming exist [15–17], they also cannot express

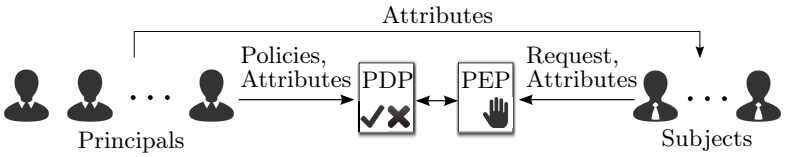

Fig. 1. The system model with the Policy Enforcement Point (PEP), Policy Decision Point (PDP), principals, subjects, requests, and attributes

all composition operators found in policy algebras, e.g. the only-one-applicable operator; that is, they are functionally incomplete.

XACML 3 is currently the only access control language supporting decentralized composite access control. Similarly to BelLog, XACML 3 has four policy decisions and operators for encoding delegation and policy composition. In contrast to BelLog, XACML is informal and some aspects are underspecified; for example, loop handling in delegation chains is left to implementations. Moreover, XACML 3 has a fixed set of composition operators and new operators cannot be added as syntactic extensions. Kolovski et al. [18] give a formalization of XACML 3 which focuses on delegations and supports only three composition operators. BelLog, in contrast, supports all finitary composition operators.

Finally, we remark that BelLog is not meant to be an all-encompassing policy specification language. For example, the constraint-based conditions of [11] are not expressible in BelLog.

Organization. In §2, we introduce our system model. In §3, we define our logic programming language BelLog and define the main decision problems for BelLog programs. In §4, we illustrate the specification of decentralized composite policies in BelLog. In §5, we present our policy analysis framework. We conclude the paper in §6. Note that proofs and technical details can be found in the extended version of the paper [19].

2 System Model and the Running Example

A Policy Decision Point (PDP) maps access requests to policy decisions and a Policy Enforcement Point (PEP) enforces the policy decisions made by the PDP. We consider an open distributed system, as illustrated in Figure 1, where there are multiple principals that may issue policies and attributes and store them at the PDP. One principal is designated as the PDP's administrator. The administrator writes the policy against which all requests are evaluated.

Subject and object attributes are issued and signed by principals. Authority over attributes can be delegated to other principals. An attribute issued by a principal is either stored at the PDP, or given to the subject, who may provide it to the PDP together with a request. Attributes that are not explicitly communicated to the PDP are assumed not to have been issued, as is the case in other decentralized systems [1]. A policy domain database contains the identifiers of objects such as roles, file names, etc. Both the administrator and authorized principals can extend this database.

To illustrate our system model, consider a grid system that stores files for multiple research projects. Each project has one or more project leaders. The grid system has one PDP that decides access for all files. The PDP's policy, inspired by policies in the Swedish Grid Initiative (SweGrid) system [2], is:

R1: A project leader controls access to the project's files and folders, and can delegate these rights.
R2: If there is a conflicting decision among the project leaders for a given request, then grant access only to requests made by the project leaders.
R3: If no policy applies to a given request, then grant the request if its target is a public project folder, otherwise deny it.
R4: Access rights are recursively extended to sub-folders.

This policy exemplifies the tight coupling between the use of delegation and composition in decentralized composite policies. The PDP must first compute the delegations for each folder according to R1, then compose the access rights for each folder according to R2 and R3, and finally extend the policy decisions to sub-folders according to R4. Note that R4 can be encoded as delegation from a parent folder to its children. Such couplings of delegation and composition idioms prevent the decentralized composite policies from being split into and evaluated as two independent, delegation and composition, parts.

3 BelLog

In this section, we define the syntax and semantics of BelLog and study the time complexity of its decision problems. BelLog builds upon the syntax and semantics of stratified Datalog [5], and extends it over a four-valued truth space. We see BelLog as a foundation for constructing high-level access control languages, and we therefore present BelLog as a generic many-valued logic programming language. In §4, we illustrate how BelLog can be used to specify practical access control policies.

Syntax. We fix a finite set $\mathcal{P}$ of predicate symbols, where $\mathcal{D}_4 = \{\mathsf{f}_4, \bot_4, \top_4, \mathsf{t}_4\} \subseteq \mathcal{P}$, along with a countably infinite set $\mathcal{C}$ of constants, and a countably infinite set $\mathcal{V}$ of variables. The sets $\mathcal{P}$, $\mathcal{C}$, and $\mathcal{V}$ are pairwise disjoint. Each predicate symbol $p \in \mathcal{P}$ is associated with an arity and we may write p^n to emphasize that p's arity is n. The predicate symbols in $\mathcal{D}_4$ have zero arity. As a convention, we write P to denote a BelLog program and use the remaining uppercase letters to denote variables. Predicate and constant symbols are written using lowercase *italic* and sans font respectively.

A *domain* Σ is a nonempty finite set of constants. We associate a domain Σ with a set of *atoms* $\mathcal{A}_{\Sigma(\mathcal{V})} = \{p^n(t_1, \cdots, t_n) \mid p^n \in \mathcal{P}, \{t_1, \cdots, t_n\} \subseteq \Sigma \cup \mathcal{V}\}$. A *literal* is either a, $\neg a$, or $\sim a$, for $a \in \mathcal{A}_{\Sigma(\mathcal{V})}$, and $\mathcal{L}_{\Sigma(\mathcal{V})}$ denotes the set of literals over Σ. We refer to $\neg a$ as *negative literals* and to a and $\sim a$ as *non-negative literals*. The function $vars : \mathcal{A}_{\Sigma(\mathcal{V})} \mapsto 2^{\mathcal{V}}$ maps atoms to the set of variables appearing in them. An atom a is *ground* iff $vars(a) = \emptyset$, and $\mathcal{A}_{\Sigma(\emptyset)}$ denotes the set of ground atoms. We extend *vars* to literals in the standard way.

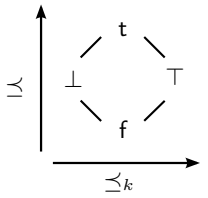

Fig. 2. BELLOG's truth space

	¬	∼
f	t	f
⊥	⊥	⊤
⊤	⊤	⊥
t	f	t

∧	f	⊥	⊤	t
f	f	f	f	f
⊥	f	⊥	f	⊥
⊤	f	f	⊤	⊤
t	f	⊥	⊤	t

∨	f	⊥	⊤	t
f	f	⊥	⊤	t
⊥	⊥	⊥	t	t
⊤	⊤	t	⊤	t
t	t	t	t	t

Fig. 3. Truth tables of BELLOG's operators

A BELLOG program, defined over the domain Σ, is a finite set of *rules* of the form:

$$p \leftarrow q_1, \ldots, q_n \ ,$$

where $n > 0$, $p \in \mathcal{A}_{\Sigma(\mathcal{V})}$, $\{q_1, \cdots, q_n\} \subseteq \mathcal{L}_{\Sigma(\mathcal{V})}$, and $vars(p) \subseteq \bigcup_{1 \leq i \leq n} vars(q_i)$. We refer to p as the rule's head and to $q_1, \ldots, q_n$ as the rule's body.

The predicate symbols in a BELLOG program P are partitioned into intensionally defined predicates, denoted idb_P, and extensionally defined predicates, denoted edb_P. The set idb_P contains all predicate symbols that appear in the heads of P's rules, and the set edb_P contains the remaining predicate symbols. We write $\mathcal{A}^{\mathsf{edb}_P}_{\Sigma(\mathcal{V})}$ ($\mathcal{L}^{\mathsf{edb}_P}_{\Sigma(\mathcal{V})}$) and $\mathcal{A}^{\mathsf{idb}_P}_{\Sigma(\mathcal{V})}$ ($\mathcal{L}^{\mathsf{idb}_P}_{\Sigma(\mathcal{V})}$) to denote the sets of atoms (literals) constructed from predicate symbols in edb_P and idb_P respectively.

A rule $p \leftarrow q_1, \cdots, q_n$ is ground iff all the literals in its body are ground. The *grounding* of a BELLOG program P is the finite set of ground rules, denoted by $P^{\downarrow}$, obtained by substituting all variables in P's rules with constants from Σ in all possible ways.

A BELLOG program P is *stratified* iff the rules in P can be partitioned into sets $P_0, \cdots, P_n$ called strata, such that: (1) for every predicate symbol p, all rules with p in their heads are in one stratum P_i; (2) if a predicate symbol p occurs as a non-negative literal in a rule of P_i, then all rules with p in their heads are in a stratum P_j with $j \leq i$; (3) if a predicate symbol p occurs as a negative literal in a rule's body in P_i, then all rules with p in their heads are in a stratum P_j with $j < i$. The given definition of stratified BELLOG extends with non-negative literals that of stratified Datalog [20].

Semantics. The truth space of BELLOG is the lattice $(\mathcal{D}, \preceq, \wedge, \vee)$, where $\mathcal{D} = \{\mathsf{f}, \bot, \top, \mathsf{t}\}$, $\preceq$ is the partial truth ordering on $\mathcal{D}$, and $\wedge$ and $\vee$ are the meet and join operators. Figure 2 shows the lattice's Hasse diagram, where $\preceq$ is depicted upwards. We adopt the meaning of the non-classical truth values $\bot$ and $\top$ from Belnap's four-valued logic [6]: $\bot$ denotes *missing information* and $\top$ denotes *conflicting information*. We define the partial knowledge ordering on $\mathcal{D}$, denoted with $\preceq_k$, and depict it in Figure 2 rightwards. We denote the meet and join operators on the lattice $(\mathcal{D}, \preceq_k)$ by $\otimes$ and $\oplus$, respectively. The truth tables of the unary operators $\neg$ and $\sim$ are given in Figure 3, where we also depict the truth tables for the operators $\wedge$ and $\vee$ for convenience.

An *interpretation* I, over a domain Σ, is a function $I : \mathcal{A}_{\Sigma(\emptyset)} \rightarrow \mathcal{D}$, mapping ground atoms to truth values, where $I(\mathsf{f}_4) = \mathsf{f}$, $I(\bot_4) = \bot$, $I(\top_4) = \top$, and $I(\mathsf{t}_4) = \mathsf{t}$. Fix a domain Σ, and let $\mathcal{I}$ be the set of all interpretations over Σ.

We define a partial ordering $\sqsubseteq$ on interpretations: given $I_1, I_2 \in \mathcal{I}$, $I_1 \sqsubseteq I_2$ iff $\forall a \in \mathcal{A}_{\Sigma(\emptyset)}.\ I_1(a) \preceq I_2(a)$. We define the meet $\sqcap$ and join $\sqcup$ operators on $\mathcal{I}$ as: $I_1 \sqcap I_2 = \lambda a.\ I_1(a) \wedge I_2(a)$ and $I_1 \sqcup I_2 = \lambda a.\ I_1(a) \vee I_2(a)$. The structure $(\mathcal{I}, \sqsubseteq, \sqcap, \sqcup, I_{\mathsf{f}}, I_{\mathsf{t}})$ is a complete lattice where $I_{\mathsf{f}} = \lambda a.\mathsf{f}$ is the least element and $I_{\mathsf{t}} = \lambda a.\mathsf{t}$ is the greatest element. Given a continuous function $\Phi : \mathcal{I} \to \mathcal{I}$, we write $\lceil \Phi \rceil$ for the least fixed point of Φ. The interpretation $\lceil \Phi \rceil$ is calculated, using the Kleene fixed point theorem, as M^ω where $M^0 = I_{\mathsf{f}}$, and $M^{i+1} = \Phi(M^i)$ for $i \geq 0$.

We extend interpretations over the operators $\neg$ and $\sim$ as $I(\neg a) = \neg I(a)$ and $I(\sim a) = \sim I(a)$ respectively, where $a \in \mathcal{A}_{\Sigma(\emptyset)}$. We also extend interpretations over vectors of literals as $I(\boldsymbol{l}) = I(l_1) \wedge \cdots \wedge I(l_n)$ where $\boldsymbol{l} = l_1, \cdots, l_n$ and $\{l_1, \cdots, l_n\} \subseteq \mathcal{L}_{\Sigma(\emptyset)}$. We write $\bigvee\{v_1, \cdots v_n\}$ for $v_1 \vee \cdots \vee v_n$. For the empty set we put $\bigvee\{\} = \mathsf{f}$.

An interpretation I is a *model* of a given program P iff $\forall (a \leftarrow \boldsymbol{l}) \in P^{\downarrow}.\ I(a) \succeq I(\boldsymbol{l})$. A model therefore, for every rule, assigns to the head a truth value no smaller, in $\preceq$, than the truth value assigned to the body. A model I is *supported* iff $\forall a \in \mathcal{A}_{\Sigma(\emptyset)}.\ I(a) = \bigvee\{I(\boldsymbol{l}) \mid (a \leftarrow \boldsymbol{l}) \in P^{\downarrow}\}$. Note that the definition of supported models for BelLog programs extends that of stratified Datalog. Intuitively, a model I is supported if it does not over-assign truth values to head atoms. In contrast to stratified Datalog, BelLog's truth values are not totally ordered; therefore, a supported model I of a BelLog program P does not guarantee that for an atom a there is a rule $(a \leftarrow \boldsymbol{l}) \in P^{\downarrow}$ such that $I(a) = I(\boldsymbol{l})$. For example, for the program $P = \{a \leftarrow \top_4, a \leftarrow \bot_4\}$ the interpretation $I = \{a \mapsto \mathsf{t}\}$ is a supported model; note that $\{a \mapsto \bot\}$ and $\{a \mapsto \top\}$ are not models of P.

We associate a BelLog program P with the operator $T_P : \mathcal{I} \mapsto \mathcal{I}$:

$$T_P(J)(a) = \bigvee\{J(\boldsymbol{l}) \mid (a \leftarrow \boldsymbol{l}) \in P^{\downarrow}\}$$

Lemma 1. *Given a* BelLog *program P, an interpretation I is a supported model iff $T_P(I) = I$.*

The proof follows immediately from the definition of T_P.

In general, a program P may have multiple supported models. For instance, any interpretation is a supported model for the program $\{p \leftarrow p\}$. For BelLog's semantics we choose a minimal supported model: a supported model I is *minimal* iff there does not exist another supported model I' such that $I' \sqsubset I$. For a program P where only non-negative literals are in its rules, T_P is monotone, hence continuous due to the finiteness of $\mathcal{I}$, and has a unique minimal supported model. In contrast, if a program P contains negative literals in its rules, then the operator T_P is not monotone, and there could be multiple minimal supported models. For example, the program $P = \{a \leftarrow \neg b\}$ has more than one minimal supported models, e.g. $\{a \mapsto \mathsf{f}, b \mapsto \mathsf{t}\}$ and $\{a \mapsto \mathsf{t}, b \mapsto \mathsf{f}\}$.

For a stratified BelLog program P, we construct one minimal supported model by computing, for each strata of P, the minimal supported model that contains the model of the previous stratum. This construction is analogous to that of stratified Datalog given in [21]. To define the model construction, we introduce the following notation. We write $(P^{\downarrow}) \triangleleft I$ for the program obtained by

replacing all literals in $P^{\downarrow}$ constructed with edb_P predicate symbols with their truth values according to I. Formally,

$$(P^{\downarrow}) \triangleleft I \;=\; \{p \leftarrow q'_1, \cdots, q'_n \;\mid (p \leftarrow q_1, \cdots, q_n) \in P^{\downarrow}, \\ q'_i = I(q_i) \text{ if } q_i \in \mathcal{L}^{\mathsf{edb}_P}_{\Sigma(\emptyset)}, \text{otherwise } q'_i = q_i\} \; .$$

Note that all negative literals in a stratum P_i of a stratified BelLog program are constructed with predicate symbols in edb_{P_i}. Given an interpretation I, the program $P_i^{\downarrow} \triangleleft I$ therefore contains only non-negative literals, and the operator $T_{P_i^{\downarrow} \triangleleft I}$ is monotone.

We now define the model semantics of a stratified BelLog program:

Definition 1. *Given a stratified* BelLog *program P, with strata $P_0, \cdots, P_n$, the model of P, denoted $[\![P]\!]$, is the interpretation M_n, where $M_{-1} = I_{\mathsf{f}}$, and $M_i = \lceil T_{P_i^{\downarrow} \triangleleft M_{i-1}} \rceil \sqcup M_{i-1}$ for $0 \leq i \leq n$.*

Each M_i, for $0 \leq i \leq n$, is well-defined because the operators $T_{P_i^{\downarrow} \triangleleft M_{i-1}}$ are monotone, and therefore continuous because the lattice $(\mathcal{I}, \sqsubseteq, \sqcap, \sqcup)$ is finite.

Theorem 1. *Given a stratified* BelLog *program P, $[\![P]\!]$ is a minimal supported model.*

For the previous example $P = \{a \leftarrow \neg b\}$, the given construction results in $[\![P]\!] = \{a \mapsto \mathsf{t}, b \mapsto \mathsf{f}\}$. For details on our choice of semantics see [19].

We remark that a BelLog program P that does not use the predicates $\top_4$, $\bot_4$, and the operator $\sim$ in its rules is a syntactically valid stratified Datalog program. Furthermore, stratified BelLog subsumes stratified Datalog; see [19]. In particular, this means that BelLog can express all policy languages based on stratified Datalog.

The *input* to a BelLog program P is an interpretation $I \in \mathcal{I}$, where all atoms from $\mathcal{A}^{\mathsf{idb}_P}_{\Sigma(\emptyset)}$ are mapped to f. For a program P and the input I, we write $[\![P]\!]_I$ as a shorthand for $[\![P \cup P']\!]$, where $P' = \{a \leftarrow v_4 \;\mid\; I(a) = v\}$ and $v \in \mathcal{D}$.

From the definition of stratification, it is immediate that given a stratified program P with strata $P_0, \cdots, P_n$, and an input I, the program $P \cup P'$ can be stratified into strata $P', P_0, \cdots, P_n$.

We finally remark that the semantics of a BelLog program is independent of the given stratification. The proof can be found in [19].

Decision Problems. We define BelLog's decision problems. In §5, we reduce the decision problems within our policy analysis framework to BelLog's decision problems.

Let P be a stratified BelLog program, Σ be a domain of constants, and q be a ground atom. For a given input I, the *query entailment* decision problem, denoted $P \models^I_\Sigma q$, asks whether $[\![P]\!]_I(q) = \mathsf{t}$. The general case of $[\![P]\!]_I(q) = v$, with $v \in \mathcal{D}$, is immediately reducible to the query entailment problem. The *query validity* decision problem, denoted $P \models_\Sigma q$, asks whether for all inputs I defined over Σ, $P \models^I_\Sigma q$. Similarly to the *data* complexity of Datalog [22], we study the complexity of the given decision problems when the maximum arity of

predicates in P and the set of variables that appear in P are fixed. The input size for BelLog's decision problems is thus determined by the number of predicate symbols in $\mathcal{P}$, the number of rules in P, and the number of constants in the domain Σ.

Theorem 2. *The query entailment problem and the query validity problem belong, respectively, to the complexity classes* PTIME *and* CO-NP-COMPLETE.

We next consider a generalization of the query validity problem. Let Σ_P denote the set of constants that appear in P. The *all-domains query validity* decision problem, denoted $P \models q$, asks whether $P \models_{\Sigma'} q$ for all domains $\Sigma' \subseteq \mathcal{C}$ that contain Σ_P and the constants in q; recall that $\mathcal{C}$ is the infinite set of constants. The problem of all-domains query validity is in general undecidable for BelLog programs, because the problem of query validity in Datalog, which is undecidable [23], can be reduced to this problem. We show, however, that all-domains query validity is decidable for any stratified BelLog program P that has only unary predicate symbols in edb_P. We call those *unary-edb programs*. We show in §5 that the unary-edb BelLog programs capture a useful class of policies. Namely, those policies where the set of principals is finite.

Theorem 3. *The all-domains query validity problem for a unary-edb* BelLog *program belongs to the complexity class* CO-NEXP.

Note that the input for the all-domains query validity problem is determined only by the number of predicate symbols in $\mathcal{P}$ and the number of rules in the program P.

Syntactic Extensions. We now present a set of syntactic extension to BelLog to ease the specification of complex rules. In §4, we use them for writing decentralized composite policies.

We extend the syntax for writing policy rules to

$$\begin{aligned} \mathit{rule} &::= p \leftarrow \mathit{body} \\ \mathit{body} &::= q_1, \cdots, q_n \mid \neg \mathit{body} \mid {\sim}\mathit{body} \mid \mathit{body} \wedge \mathit{body} \ , \end{aligned}$$

where $n > 0$, $p \in \mathcal{A}_{\Sigma(\mathcal{V})}$, and $\{q_1, \cdots, q_n\} \subseteq \mathcal{L}_{\Sigma(\mathcal{V})}$. We call the rules of the form $p \leftarrow q_1, \cdots, q_n$ *basic rules* and the remaining rules *composite rules*. Similarly to basic rules, we require that for any composite rule $p \leftarrow \mathit{body}$, $\mathit{vars}(p) \subseteq \mathit{vars}(\mathit{body})$.

We define the translation function $\mathcal{T}$ that maps a basic rule r to the set $\{r\}$:

$$\mathcal{T}(p \leftarrow q_1, \cdots, q_n) = \{p \leftarrow q_1, \cdots, q_n\} \ ,$$

and maps a composite rule $p \leftarrow \mathit{body}$ to a set of basic rules:

$$\begin{aligned} \mathcal{T}(p \leftarrow \neg \mathit{body}) &= \{p \leftarrow \neg p_{\mathsf{fresh}}(\boldsymbol{X})\} \cup \mathcal{T}(p_{\mathsf{fresh}}(\boldsymbol{X}) \leftarrow \mathit{body}) \\ \mathcal{T}(p \leftarrow {\sim}\mathit{body}) &= \{p \leftarrow {\sim} p_{\mathsf{fresh}}(\boldsymbol{X})\} \cup \mathcal{T}(p_{\mathsf{fresh}}(\boldsymbol{X}) \leftarrow \mathit{body}) \\ \mathcal{T}(p \leftarrow \mathit{body}_1 \wedge \mathit{body}_2) &= \{p \leftarrow p_{\mathsf{fresh1}}(\boldsymbol{X}_1), p_{\mathsf{fresh2}}(\boldsymbol{X}_2)\} \\ &\quad \cup \mathcal{T}(p_{\mathsf{fresh1}}(\boldsymbol{X}_1) \leftarrow \mathit{body}_1) \cup \mathcal{T}(p_{\mathsf{fresh2}}(\boldsymbol{X}_2) \leftarrow \mathit{body}_2) \end{aligned}$$

$$p \vee q := \neg(\neg p \wedge \neg q) \qquad p \otimes q := (p \wedge \bot) \vee (q \wedge \bot) \vee (p \wedge q)$$
$$p \oplus q := (p \wedge \top) \vee (q \wedge \top) \vee (p \wedge q) \qquad p = \mathsf{t} := p \wedge {\sim}p$$
$$p = \mathsf{f} := \neg(p \vee {\sim}p) \qquad p = \bot := (p \neq \mathsf{f}) \wedge (p \neq \mathsf{t}) \wedge ((p \vee \top) = \mathsf{t})$$
$$p = \top := (p \neq \mathsf{f}) \wedge (p \neq \mathsf{t}) \wedge ((p \vee \bot) = \mathsf{t}) \qquad p \neq v := \neg(p = v)$$

Fig. 4. Derived connectives for combining composite rule bodies. Here p, q, and c denote rule bodies and $v \in \mathcal{D}$.

In these rules p_{fresh}, p_{fresh1}, p_{fresh2} are predicate symbols that do not appear in $\mathcal{P}$, $\boldsymbol{X} = \mathit{vars}(\mathit{body})$ and $\boldsymbol{X}_i = \mathit{vars}(\mathit{body}_i)$ for $i \in \{1, 2\}$. Note that the recursive function $\mathcal{T}$ terminates for any composite rule and yields a set of basic rules; see [19]. The size of the set is linear in the number of nested *bodies* in the composite rule.

The meaning of a BelLog program P with composite rules is that of the BelLog program $P' = \bigcup_{r \in P}(\mathcal{T}(r))$. For example, consider the composite rule:

$$p(X) \leftarrow \neg{\sim}q(X, Y) \ .$$

The function $\mathcal{T}$ translates this composite rule into a set of basic rules:

$$\{p(X) \leftarrow \neg p_{\mathsf{fresh}}(X, Y),\ p_{\mathsf{fresh}}(X, Y) \leftarrow {\sim}q(X, Y)\} \ .$$

A BelLog program P with composite rules is *well-formed* iff its rules can be partitioned into sets $P_0, \cdots, P_n$ such that: (1) for every predicate symbol p, all rules with p in their heads are in one stratum P_i; (2) if a predicate symbol p occurs as a non-negative literal in a basic body in P_i, then all rules with p in their heads are in a stratum P_j with $j \leq i$; and (3) if a predicate symbol p occurs in the body of a composite rule in P_i or as a negative literal in a basic rule in P_i, then all rules with p in their heads are in a stratum P_j with $j < i$. Note that well-formed BelLog extends stratified BelLog with the condition that if a predicate symbol p occurs in the body of a composite rule in P_i, then all rules with p in their heads are in a stratum P_j with $j < i$. This is a sufficient but not necessary condition that any composite rule of a well-formed program is translated into a stratified set of basic rules.

Theorem 4. *The translation of a well-formed* BelLog *program with composite rules is a stratified* BelLog *program.*

In Figure 4, we derive additional connectives using syntactic combinations of $\neg$, $\sim$, and $\wedge$. The binary connective $\_ \vee \_$ corresponds to the join operator on the lattice $(\mathcal{D}, \preceq)$, and the binary connectives $\_ \otimes \_$ and $\_ \oplus \_$ correspond to the meet and join operators on the lattice $(\mathcal{D}, \preceq_k)$, respectively; for details see [6]. The unary connective $\_ = v$, where $v \in \mathcal{D}$, indicates whether the truth value assigned to the atom is v. The result of $p = v$ is t if p's result is v, and f otherwise. The composition $p \neq v$ returns t only if p's result is not v, otherwise it returns f. Furthermore, we formally establish that BelLog can represent any n-ary operator $D^n \to D$:

Theorem 5. *Given an operator* $g : D^n \to D$ *and a list of* n *rule bodies* $q_1, \cdots, q_n$*, there exists a* body *expression* ϕ *for a* BelLog *composite rule* $p \leftarrow \phi$ *such that*

$$[\![P]\!]_I(p) = g([\![P]\!]_I(q_1), \ldots, [\![P]\!]_I(q_n)) \ ,$$

for all inputs I*, and programs* P *where* $\{p \leftarrow \phi\} \subseteq P$ *and* p *is not the head of any other rule.*

4 Decentralized Composite Policies in BelLog

We first introduce the basic building blocks, namely attributes and delegations, and then we demonstrate how to encode decentralized composite policies in BelLog, including the grid policy from §2. We conclude with a discussion of BelLog's more intricate features for policy specifications.

We assume that the PDP's domain database contains all constants that appear in the policies, attributes, and access requests, as well as any other additional constants which may denote roles, file names, etc.

Attributes and Delegations. We represent attributes with $attribute\_name(\cdot)$ predicate symbols. We take the first argument of an attribute as the issuing principal's identifier. For example, $hr(\mathsf{ann}, \mathsf{fred})$ denotes that, according to Ann, Fred works in the Human Resources department. To highlight the attribute's issuer, we may write $hr(\mathsf{fred})@\mathsf{ann}$ instead of $hr(\mathsf{ann}, \mathsf{fred})$.

The truth value of an attribute a is t if it is either stored at the PDP or provided by the subject; otherwise it is f. In short, the attributes are by default assumed not to exist if they are not present. For some policies it may however be more appropriate to assume that a given attribute (e.g. an attribute that is provided by the subject) is missing ($\bot$) rather than non-existent (f). BelLog can accommodate for such policies too. For example, given an attribute a, we can define its *assume-missing* counterpart $a_\bot$ with the rule $a_\bot \leftarrow a \vee \bot$.

Attribute delegations are specified with BelLog rules where the rule's head is the delegated attribute and the rule body is the delegation condition. For example, with the rule

$$researcher(S)@\mathsf{ann} \leftarrow hr(S')@\mathsf{ann}, labcard(S)@S' \ ,$$

Ann asserts that a subject S is a researcher if a subject S' with the attribute hr asserts that S is a researcher. That is, Ann delegates the attribute *researcher* to subjects that have the attribute hr. For example, if Fred has the attribute hr and issues $labcard(\mathsf{dave})@\mathsf{fred}$, then the PDP derives $researcher(\mathsf{dave})@\mathsf{ann}$.

Delegations may require non-monotonic operators. Imagine that Ann stores at the PDP a list of revoked subjects, and she will not accept delegations of the attribute *researcher* for revoked subjects. We extend her delegation rule as

$$researcher(S)@\mathsf{ann} \leftarrow hr(S')@\mathsf{ann}, labcard(S)@S', \neg revoked(S)@\mathsf{ann} \ .$$

Non-monotonic operators must be used with caution when applied to the attributes that subjects supply. This is because a subject may gain access if she can withhold the attribute *revoked* from the PDP; cf. [8]. In §5, we return to this

issue and show how one can verify whether a policy is monotone with respect to the attributes provided by the subject.

BelLog's composite rules can be used to express more complex delegation conditions. In our grid example, the administrator may for instance require two project leaders—Ann and Fred—to agree on the *pub* file attribute, denoting that a file is public. This is written as

$$pub\_agree(F)@\mathsf{admin} \leftarrow pub(F)@\mathsf{ann} \oplus pub(F)@\mathsf{fred}\ ,$$

where $\oplus$ is the maximal agreement operator. Note that the administrator derives a conflict if the principals disagree whether a file is public, because $\mathsf{f} \oplus \mathsf{t} = \top$.

As illustrated, BelLog can specify standard attribute delegations, as well as non-monotonic delegation idioms which cannot be captured in existing Datalog-based languages. There are other delegation idioms that BelLog can express, but we omit their presentation due to space constraints. For example, the hand-off idiom [14], where a principal delegates authority over all attributes, can be expressed in BelLog by representing attributes with a predicate *says* where one of the arguments denotes an attribute name.

Policy Decisions. We take the $\mathsf{t}, \mathsf{f}, \bot$, and $\top$ elements as, respectively, *grant*, *deny*, *gap*, and *conflict* policy decisions. The *gap* decision indicates that a policy neither grants nor denies a request, and *conflict* indicates that a policy can both grant and deny a request. The partial ordering $\preceq$ in Figure 2 defines the *permissiveness* of policy decisions. The meet $\wedge$ and join $\vee$ operators on the lattice $(\mathcal{D}, \preceq)$ correspond to the standard *deny-override* and *permit-override* operators for composing policy decisions. The meet $\otimes$ and join $\oplus$ operators on the lattice $(\mathcal{D}, \preceq_k)$ correspond to the *maximal agreement* and *minimal agreement* composition operators; see [15].

Policies. A principal can issue multiple policies for different subjects and resources; we insist however that each principal has one designated root policy. A root policy combines all of the principal's sub-policies and possibly other principals' policies. In our grid scenario, we use the atom $pol\_name(Sub, File)@Prin$ to denote the decision of the policy *name*, issued by *Prin*, for *Sub* accessing *File*. We fix the atom $pol(Sub, File)@Prin$ to denote *Prin*'s root policy. For example, when the PDP derives t for the atom $pol(\mathsf{fred}, \mathsf{foo.txt})@\mathsf{piet}$, the PDP interprets this as "Piet's root policy grants Fred access to the file foo.txt". Principals may choose any other predicate symbols to denote decisions of their sub-policies.

Policies are encoded as BelLog rules where the head of a policy rule is a policy name atom. For example, the project leader Piet may issue the policy

$$pol(S, F)@\mathsf{piet} \leftarrow researcher(S)@\mathsf{piet}, prj\_file(F)@\mathsf{piet}\ ,$$

which grants his researchers S access to any project files F. Similarly, Ann, who is a project leader, may issue the policy

$$\begin{aligned} pol(\mathsf{ann}, F)@\mathsf{ann} &\leftarrow prj\_file(F)@\mathsf{ann} \\ pol(S, F)@\mathsf{ann} &\leftarrow pol(S', F)@\mathsf{ann}, give\_access(S, F)@S'\ , \end{aligned}$$

$$p \triangleleft c \triangleright q := ((c = \mathsf{t}) \wedge p) \vee ((c \neq \mathsf{t}) \wedge q) \qquad p \overset{v}{\mapsto} q := q \triangleleft (p = v) \triangleright p$$
$$p \bowtie q := p \triangleleft (q = \bot) \triangleright (q \triangleleft (p = \bot) \triangleright \bot) \qquad p \blacktriangleright q := q \triangleleft (p = \mathsf{t}) \triangleright \bot$$

Fig. 5. Conditional and override policy composition operators

where the first rule grants Ann access to any project file F, and the second rule states that any subject S' with access to F may delegate this access to any subject S by issuing a *give_access* attribute. Then, Ann may provide access to Fred by issuing $give\_access(\mathsf{fred}, \mathsf{foo.txt})@\mathsf{ann}$; Fred too may issue $give\_access(\mathsf{dave}, \mathsf{foo.txt})@\mathsf{fred}$ to further delegate to Dave access to foo.txt.

A policy can also combine the decisions of a set of sub-policies; we call these *composite* policies. A composite policy encoded with a basic BelLog rule, for example, implicitly combines the sub-policies' decisions using the deny-override $\wedge$ operator. Composite policies that combine their sub-policies' decisions with more complex composition operators, such as the gap- and conflict-override operators, are encoded with BelLog composite rules.

In addition to $\wedge$, BelLog's operators $\neg$, $\sim$, $\vee$, $\otimes$, $\oplus$ can also be employed as composition operators. To complement these operators, in Figure 5 we define further conditional and override operators for composing policies. The ternary operator $\_ \triangleleft \_ \triangleright \_$ is the *if-then-else* operator. The result of the composition $p \triangleleft c \triangleright q$ is p's decision only if c's result is t, otherwise q's decision is taken.

The binary operator $\_ \overset{v}{\mapsto} \_$ represents the *v-override operator*, where $v \in \mathcal{D}$. The result of the composition $p \overset{v}{\mapsto} q$ is q if p's decision is v, otherwise it results in p's decision. The operators $\overset{\bot}{\mapsto}$ and $\overset{\top}{\mapsto}$ correspond to the *gap-override* and *conflict-override* operators, respectively. Given a list of policies $p_1, \cdots, p_n$, we encode the operator *first-applicable* as $p_1 \overset{\bot}{\mapsto} (p_2 \overset{\bot}{\mapsto} (\cdots \overset{\bot}{\mapsto} p_n))$, i.e. the composition takes the decision of the first policy in the list whose decision is not $\bot$.

The binary operator $\_ \bowtie \_$ is the *only-one-applicable* operator, i.e. the composition $p \bowtie q$ results in $\bot$ if both policy decisions are not $\bot$ or both decisions are $\bot$, otherwise the result is the policy decision that is not $\bot$.

The binary operator $\_ \blacktriangleright \_$ is the *on-permit-apply-second*[1] operator. The composition $p \blacktriangleright q$ returns q only if the decision of p is t, otherwise it returns $\bot$. The operator $\blacktriangleright$ is useful for specifying policies that either (1) grant or provide no decision, or (2) deny or provide no decision. For example, the policy $researcher(Sub) \blacktriangleright \mathsf{t}$ grants access only if the subject *Sub* is a researcher; otherwise, the policy returns $\bot$. In contrast, the policy $revoked(Sub) \blacktriangleright \mathsf{f}$ denies access if the subject *Sub* is revoked, and provides no decision otherwise. We also use the operator $\blacktriangleright$ for specifying policies with policy targets, which define the requests that are applicable to a policy. Given a policy p and its target p_{target}, $p_{\mathsf{target}} \blacktriangleright p$ results in $\bot$ if p_{target} does not evaluate to t, otherwise it results in p's decision.

[1] The on-permit-apply-second operator has been recently proposed as an additional operator for the XACML 3 standard. See [24] for full description.

We finally remark that BELLOG can express any four-valued policy composition language, such as PBel [7]. This is a corollary of Theorem 5.

Grid Policy. We now exercise these operators in our grid scenario. The administrator may compose the policies issued by the project leaders Piet and Ann with the maximal agreement operator:

$$pol\_leaders(S,F)@\mathsf{admin} \leftarrow pol(S,F)@\mathsf{piet} \oplus pol(S,F)@\mathsf{ann}\ .$$

For brevity, we have not specified the policies of Piet and Ann. The composition of their policies may result in conflicts and gaps. According to requirements R2 and R3 (see §2), the administrator must resolve conflicts by granting requests made by project leaders, and resolve gaps by granting access only to public folders. The *pol_root* policy encodes these requirements:

$$pol\_root(S,F)@\mathsf{admin} \leftarrow$$
$$(pol\_leaders(S,F)@\mathsf{admin} \overset{\top}{\mapsto} prj\_leader(S)@\mathsf{admin}) \overset{\bot}{\mapsto} pub(F)@\mathsf{admin}\ .$$

The composite policy *pol_leaders* considers the decisions of Piet's and Ann's policies for all requests. The administrator may, however, want to consider the decisions of Piet's policy only for the files contained in the folder prj1. This can be encoded by defining a policy with an explicit policy target:

$$pol\_piet(S,F)@\mathsf{admin} \leftarrow contains(\mathsf{prj1},F)@\mathsf{admin} \blacktriangleright pol(S,F)@\mathsf{piet}\ ,$$

where the attribute $contains(F_1,F_2)@\mathsf{admin}$ indicates that the folder F_1 contains F_2. The attribute is transitively assigned to sub-folders:

$$contains(F_1,F_2)@\mathsf{admin} \leftarrow subfolder(F_1,F_2)@\mathsf{fs}\ ,$$
$$contains(F_1,F_3)@\mathsf{admin} \leftarrow contains(F_1,F_2)@\mathsf{admin}, contains(F_2,F_3)@\mathsf{admin}\ ,$$

where the attribute $subfolder(F_1,F_2)@\mathsf{fs}$ is provided by the file system fs and indicates that F_1 is directly contained in F_2. Note that the policy *pol_piet* results in $\bot$ for any request to a file not contained in the folder prj1.

The administrator must also encode the requirement R4, which states that any access right to a folder is transitively extended to sub-folders. Namely

$$pol\_root(S,F)@\mathsf{admin} \leftarrow contains(F',F)@\mathsf{admin}, pol\_root(S,F')@\mathsf{admin}\ .$$

Note that the policy decision for a folder is extended to sub-folders with the permit-override operator. This is because instantiating the variable F' results in multiple rules with the same head atom, which are combined with the operator $\vee$ according to BELLOG's semantics. To illustrate this, consider the folder f_3, where f_3 is contained in f_2, which in turn is contained in f_1. Instantiating the variable F' and simplifying the instantiated rules result in the following rule:

$$pol\_root(S,f_3)@\mathsf{admin} \leftarrow pol\_root(S,f_1)@\mathsf{admin} \vee pol\_root(S,f_2)@\mathsf{admin}\ .$$

Alternatively, the administrator may want to combine the instantiated rule bodies with deny-override, maximal agreement, or minimal agreement. We show how this can be done with BELLOG's intensional operators, defined below.

Intensional Compositions. So far, we have presented *extensional* policy composition operators that compose a fixed, explicitly given list of sub-policies. For example, we used

$$pol\_leaders(S,F)@\mathsf{admin} \leftarrow pol(S,F)@\mathsf{piet} \oplus pol(S,F)@\mathsf{ann}$$

to combine policies of two project leaders, one from Piet and one from Ann, with the maximal agreement operator. Such extensional encodings are tediously "static", because if new project leaders are added to or removed from the PDP, then the administrator must explicitly change the policy rule. Alternatively, the administrator may write a rule that composes the policies that are issued by any principal who is a project leader. One attempt to do this is:

$$pol\_leaders(S,F)@\mathsf{admin} \leftarrow pol(S,F)@P, prj\_leader(P)@\mathsf{admin}\ ,$$

where the set of composed policies is *intensionally* defined as those issued by project leaders. This attempt however fails because the project leaders' policies are implicitly combined with the permit-override operator, instead of the maximal agreement operator $\oplus$. This is because BelLog's semantics, much like other logic programs, uses the join operator $\vee$ when combining rule bodies with the same head atom.

We extend BelLog's syntax with additional operators to account for intensional compositions:

$$rule ::= p \leftarrow [\ \bigvee \mid \bigwedge \mid \bigoplus \mid \bigotimes\ ]\ body\ ,$$

where $p \in \mathcal{A}_{\Sigma(V)}$, *body* is a composite rule body, as defined in §3, and $vars(p) \subseteq vars(body)$. We refer to the operators written in front of *body* as *intensional* composition operators. Intuitively, the intensional operator $\bigoplus$ combines all grounded bodies of rules with the same head atom with the $\oplus$ operator. For example, grounding the simple rule $p(\mathsf{a}) \leftarrow \bigoplus q(X)$ over the domain $\Sigma = \{\mathsf{a}, \mathsf{b}\}$ results in two grounded bodies, $q(\mathsf{a})$ and $q(\mathsf{b})$, with the same head atom $p(\mathsf{a})$. The grounded bodies are combined with $\oplus$; the meaning of $p(\mathsf{a}) \leftarrow \bigoplus q(X)$ is therefore $p(\mathsf{a}) \leftarrow q(\mathsf{a}) \oplus q(\mathsf{b})$. Other operators behave similarly with respect to their syntactic counterparts. The formal translation of the intensional operators to BelLog's core syntax is given in [19]. We remark that the intensional operators $\bigwedge, \bigoplus$, and $\bigotimes$ cannot have the head atom appear in the rule body because their encoding uses composite rules.

We can now encode the intensional composition of the project leaders' policies with the maximal agreement operator as

$$pol\_leaders(S,F)@\mathsf{admin} \leftarrow \bigoplus(pol(S,F)@P \ \triangleleft\ prj\_leader(P)@\mathsf{admin}\ \triangleright\ \bot)\ .$$

Note that the policies that are *not* issued by a project leader are replaced with $\bot$, and the composition "ignores" such policies, because $v \oplus \bot = v$ for any $v \in \mathcal{D}$.

Intensional compositions are also useful for specifying policies that propagate policy decisions over hierarchically structured data, such as file systems, role hierarchies, etc. To illustrate, we extend our grid example with Piet's policy that by default permits a subject S to access a folder F, unless Piet issues

the attribute $deny(S, F)$. In contrast to the requirement R4, he uses the deny-override operator to propagate deny decisions over the sub-folders:

$$pol\_fold(S, F)@\mathsf{piet} \leftarrow \neg deny(S, F)@\mathsf{piet}$$

$$pol(S, F)@\mathsf{piet} \leftarrow \bigwedge(pol\_fold(S, F')@\mathsf{piet} \triangleleft\ contains(F', F)@\mathsf{admin} \ \triangleright\ \mathsf{t}) \ .$$

The last rule replaces the policy decisions for folders F' that do not contain F with t, since for any $v \in \mathcal{D}$ we have $v \wedge \mathsf{t} = v$.

We summarize the key difference between intensional and extensional operators as follows. The intensional operators reflect changes in the domain (e.g. addition and removal of principals, files, etc.) through changes in the policy input. The extensional operators require explicit modification of the policy rules to reflect such changes.

5 Analysis

Writing a correct policy, i.e. one that grants and denies requests as intended by the policy writer, is often challenging in practice. This is both because policies are often initially given informally and imprecisely and because the policy writer can err in their formalization. In particular, a policy writer must foresee all possible policy inputs, understand how the delegation rules, the sub-policies, and their compositions influence the policy's behavior, and verify that the policy does not exhibit any unintended decisions. As a first step towards verifying the policy's behavior, the policy writer specifies the high-level requirements as formal policy analysis questions. Second, a decision procedure is used to check, in an automated manner, whether the analysis questions are answered positively, or not.

Below we present our framework for analyzing policies written in BelLog. A *policy set* is a set of delegations and policies, which are encoded as BelLog rules and collectively define a BelLog program. Every policy set has a designated *root policy*. The decision of a policy set for a given request is the decision of the policy set's root policy. We fix the predicate $pol(Subject, Object)$ to denote a root policy's decisions. For brevity, we omit writing the issuer of policies and attributes. We use the terms *input* and *(policy) context* interchangeably.

Policy Entailment. Policy entailment answers whether a policy set entails a given permission in a given policy context.

Definition 2. *(Policy Entailment) Given a policy set P and a policy context I, P entails the request $pol(S, O)$ iff $P \models^{I}_{\Sigma} pol(S, O)$.*

Policy entailment analysis is akin to software testing in that the policy writer checks the policy set for unintended grants and denies in specific policy contexts (i.e. test scenarios). Although limited in its scope, since the policy writer must give a specific context, determining policy entailment scales with the size of the domain, unlike the policy containment problem which we define shortly. Note that policy entailment can also be used for constructing PDPs.

To illustrate policy entailment, consider the following policy set P:

$$\{\ pol(S,O) \leftarrow (pol\_leaders(S,O) \overset{\top}{\mapsto} prj\_leader(S)) \overset{\bot}{\mapsto} pub(O)\ \}\ .$$

For simplicity we do not specify the policy $pol\_leaders$. One requirement for P, which is derived from the requirement R2 given in §2, may be to deny access to subjects who are not project leaders whenever the policy $pol\_leaders$ returns a conflict. To check this property, we may ask whether the policy set entails the permission $pol(\mathsf{fred}, \mathsf{foo.txt})$ in the context:

$$I = \{pol\_leaders(\mathsf{fred}, \mathsf{foo.txt}) \mapsto \top, prj\_leader(\mathsf{fred}) \mapsto \mathsf{f}\}\ ,$$

where the remaining atoms are mapped to f. For this context the policy set does not entail the permission, as expected.

Because the guarantees provided by entailment analysis are limited to the context provided by the policy writer, the requirement may not hold for other policy contexts. For example, the given policy set P violates its requirement for

$$I' = \{pol\_leaders(\mathsf{fred}, \mathsf{foo.txt}) \mapsto \top, prj\_leader(\mathsf{fred}) \mapsto \bot, pub(\mathsf{foo.txt}) \mapsto \mathsf{t}\}\ ,$$

because the policy set entails $pol(\mathsf{fred}, \mathsf{foo.txt})$, although $pol\_leaders$ results in a conflict and the PDP does not know whether Fred is a project leader.

Deciding policy entailment is reducible to query entailment; see §3. Policy entailment can be therefore decided in time polynomial in the size of the context.

Policy Containment. Policy containment thoroughly analyzes a policy set against all policy contexts. It can be used to answer questions such as: "*Do all requests in* ***all*** *policy contexts evaluate to a conclusive policy decision, i.e. grant or deny?*" Containment analysis is done either for a particular policy domain or for all possible policy domains. In more detail, the domain policy containment answers whether a policy set P_1 is more permissive than another policy set P_2 for all policy contexts for *a given domain*. The all-domains policy containment answers whether a policy set P_1 is more permissive than another policy set P_2 for all policy contexts for *all possible domains*. Even though all-domains evaluations imply those for one domain, checking for all domains is decidable only for a fragment of BelLog, as we later show.

Many analysis questions require that only specific subsets of policy contexts and requests are considered for comparisons. For example, to verify that the policy set P correctly encodes our requirement derived from R2, the policy writer may ask whether P denies all requests made by subjects who are not project leaders, for all contexts where the policy $pol\_leaders$ results in a conflict. We encode such analysis questions with a condition that constraints the contexts and requests where the policy sets are compared. Formally, the syntax for writing containment questions is

$$cond \Rightarrow P_1 \preceq P_2\ .$$

The symbols P_1 and P_2 are policy sets and $cond$ is inductively defined as

$$\begin{aligned} cond &::= \forall X.cond \mid attr \preceq v \mid v \preceq attr \mid \neg cond \mid cond \wedge cond \mid \mathsf{t} \\ v &::= \bot \mid \top\ , \end{aligned}$$

where $X \in \mathcal{V}$, $attr \in \mathcal{A}^{\mathsf{edb}_P}_{\Sigma(\mathcal{V})}$, i.e. $attr$ is an input attribute. Note that the attributes in a condition may contain variables. We write $fv(cond)$ for the set of variables in $cond$ that are not in the scope of $\forall$. We fix the variables S and O to denote the subject and the object in the request $pol(S, O)$. A policy containment question $cond \Rightarrow P_1 \preceq P_2$ is well-formed iff $fv(cond) \subseteq \{S, O\}$.

We define the satisfaction relation $\Vdash_\Sigma$ between a policy context I, a condition $cond$ of a well-formed policy containment question, and a policy domain Σ:

$$
\begin{array}{lll}
I \Vdash_\Sigma \mathsf{t} & & \\
I \Vdash_\Sigma q \preceq v & \textit{if} & I(q) \preceq v \\
I \Vdash_\Sigma v \preceq q & \textit{if} & v \preceq I(q) \\
I \Vdash_\Sigma \neg cond & \textit{if} & I \nVdash_\Sigma cond \\
I \Vdash_\Sigma cond_1 \wedge cond_2 & \textit{if} & I \Vdash_\Sigma cond_1 \textit{ and } I \Vdash_\Sigma cond_2 \\
I \Vdash_\Sigma \forall X.cond(X) & \textit{if} & \forall X \in \Sigma.\ I \Vdash_\Sigma cond(X)
\end{array}
$$

As a shorthand, in the following we write $q = v$ for $(q \preceq v) \wedge (v \preceq q)$ where $v \in \{\bot, \top\}$, $q = \mathsf{f}$ for $(q \preceq \bot) \wedge (q \preceq \top)$, and $q = \mathsf{t}$ for $\neg(q \preceq \bot) \wedge \neg(q \preceq \top)$. Given two conditions c_1 and c_2 we define their disjunction $c_1 \vee c_2$ in the standard way as $\neg(\neg c_1 \wedge \neg c_2)$. To compare the truth values of any two attributes p and q, we write $p = q$ as a shorthand for $(p = \mathsf{f} \wedge q = \mathsf{f}) \vee (p = \bot \wedge q = \bot) \vee (p = \top \wedge q = \top) \vee (p = \mathsf{t} \wedge q = \mathsf{t})$.

Definition 3. *(Domain Policy Containment) Given a question $cond \Rightarrow P_1 \preceq P_2$, and a domain Σ, then P_1 is contained in P_2 for all policy contexts over Σ that satisfy $cond$, denoted by $\Vdash_\Sigma cond \Rightarrow P_1 \preceq P_2$, iff*

$$\forall I \in \mathcal{I}, \forall S, O \in \Sigma.\ (I \Vdash_\Sigma cond) \rightarrow (\llbracket P_1 \rrbracket_I(pol(S, O)) \preceq \llbracket P_2 \rrbracket_I(pol(S, O)))\ ,$$

where $\mathcal{I}$ is the set of all policy contexts defined over the domain Σ.

Note that we overload the relation $\Vdash_\Sigma$.

In practice, the policy domain may change over time, e.g. subjects and objects are added to and removed from the system. After changes to Σ, domain policy containment may no longer hold. As mentioned, a stronger policy containment guarantee is thus to verify that P_1 is contained in P_2 for *all* domains Σ'.

Definition 4. *(All-domains Policy Containment) Given a question $cond \Rightarrow P_1 \preceq P_2$, P_1 is contained in P_2 for all policy contexts in all policy domains, denoted $\Vdash cond \Rightarrow P_1 \preceq P_2$, iff $\Vdash_\Sigma cond \Rightarrow P_1 \preceq P_2$ holds for all domains Σ.*

To illustrate how containment questions are specified and used, we start with the previously given question: "*Do all requests in **all** policy contexts evaluate to a conclusive policy decision*". To encode this question for the policy set P, we construct a policy set P' by first renaming the predicate symbol pol in P to pol' and then adding the rule

$$pol(S, O) \leftarrow (pol'(S, O) \overset{\top}{\mapsto} \mathsf{f}) \overset{\bot}{\mapsto} \mathsf{f}\ .$$

By construction, the policy set P' denies all requests that are evaluated to gap or conflict by the policy set P. Therefore, $\models_\Sigma \mathsf{t} \Rightarrow P \preceq P'$ holds iff the policy

set P is conclusive. We set the condition to t because we must check containment for all requests and for all policy contexts.

As a second example, we use policy containment to encode the requirement that the policy set P denies access to subjects who are not project leaders whenever the policy *pol_leaders* results in a conflict:

$$(pol\_leaders(S,O) = \top) \wedge \neg(prj\_leader(S) = \mathsf{t}) \Rightarrow P \preceq P_{\mathsf{f}} \ ,$$

where P_{f} is the policy set that denies all requests. This asks whether P denies $pol(S,O)$ in all contexts where the policy *pol_leaders* results in a conflict for the request $pol(S,O)$ ($pol\_leaders(S,O) = \top$) and the subject S is not a project leader ($\neg(prj\_leader(S) = \mathsf{t})$). Both domain and all-domains containment evaluations give negative answers; see the counterexample above. The policy set, however, satisfies the requirement if the attribute *prj_leader* is either t or f. We can easily encode this assumption as

$$(pol\_leaders(S,O) = \top) \wedge (prj\_leader(S) = \mathsf{f}) \Rightarrow P \preceq P_{\mathsf{f}} \ .$$

Domain and all-domains containment evaluations answer this question positively.

Policy containment is also useful for comparing a policy set's behavior in one context to its behavior in a different policy context. Consider a scenario where a subject can push some attributes to the PDP. An important property for the policy set is that a subject cannot influence the policy set to grant a request by withholding attributes. We refer to such policy sets as *push-monotonic*: whenever a subject provides fewer attributes to the PDP, the policy set results in a less permissive decision. Consider the policy set P:

$$\begin{aligned} \{ \ & pol(S,O) \leftarrow researcher(S), prj\_file(O) \\ & researcher(S) \leftarrow hr(S'), labcard(S',S), \neg revoked(S) \ \} \end{aligned}$$

The policy writer may formulate the question: "*Is the policy set more restrictive when the subject provides fewer (pushed) attributes?*" To answer this question, one must compare the policy set to itself in all policy contexts that are identical except for the attributes pushed by the subject. To encode this question, we first construct a policy set P' by renaming every predicate symbol p that appears in edb_P to p', where $\mathsf{edb}_P = \{revoked(\cdot), labcard(\cdot,\cdot), hr(\cdot), revoked(\cdot), prj\_file(\cdot)\}$. Suppose the attribute *revoked* is locally stored at the PDP and the remaining attributes are pushed by the subject. The analysis question is encoded as

$$\begin{aligned} & \forall X.\ (revoked(X) = revoked'(X)) \wedge \forall X, Y.\ (labcard(X,Y) \preceq labcard'(X,Y)) \\ & \wedge \forall X.\ (hr(X) \preceq hr'(X)) \wedge \forall X.\ (prj\_file(X) \preceq prj\_file'(X)) \Rightarrow P \preceq P' \ . \end{aligned}$$

This analysis problem asks whether P is less permissive than P' in all policy contexts that are identical for the stored attribute and all pushed attributes to P are also pushed to P'. The question indeed holds for the policy set P.

The problems of deciding domain and all-domains policy containment are reducible to domain and all-domains query validity, respectively.

Theorem 6. *Policy containment is polynomially reducible to query validity.*

Table 1. Complexity of BelLog's policy analysis problems

Analysis problem	Entailment	Domain containment	All-domains containment	All-domains containment*
Complexity	PTIME	CO-NP-COMPLETE	UNDECIDABLE	CO-NEXP

$\star$ For policies that belong to the unary-edb BelLog fragment.

Corollary 1. *The problem of domain policy containment belongs to the complexity class* CO-NP-COMPLETE. *The problem of all-domains policy containment for unary-edb policy sets belongs to the complexity class* CO-NEXP.

If a policy set has attributes associated to a single user, group, resource, etc. and there are finitely many principals, then the policy set can be written in the unary-edb fragment. This is because all attributes have the form $attr\_name(Issuer, Object)$ can be re-encoded as $attr\_name_{Issuer}(Object)$ since there are finitely many principals.

6 Conclusions

In this paper we present BelLog, a formal language for specifying access control policies that require both authority delegation and policy composition. This sets BelLog apart from the existing formal access control languages, which support either authority delegation or policy composition. BelLog can therefore specify decentralized composite policies, which thus far have lacked formal semantics; examples include policies based on the XACML 3 standard [25] and policies for large-scale distributed systems, such as [2–4,26]. We present an analysis framework for reasoning about BelLog policies and give complexity bounds for deciding policy entailment and policy containment in BelLog, summarized in Table 1.

We see BelLog as a foundation for constructing high-level policy languages for decentralized composite access control, much like Datalog is the foundation for delegation languages such as RT [12] and SecPAL [11]. We plan to build implementations of BelLog and apply them in practice. In particular we will focus on algorithms for fast evaluation of practically-relevant policies, and sound approximation techniques for deciding the policy analysis problems efficiently.

References

1. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.: The KeyNote Trust-Management System Version 2. RFC 2704 (Informational) (September 1999)
2. SNIC: SweGrid: e-Infrastructure for Computing and Storage, `http://www.snic.vr.se/projects/swegrid/`
3. Axiomatics: Policy Decision Points (September 2013)
4. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A View of Cloud Computing. Commun. ACM 53(4), 50–58 (2010)
5. Ceri, S., Gottlob, G., Tanca, L.: What You Always Wanted to Know About Datalog (And Never Dared to Ask). IEEE Trans. on Knowl. and Data Eng., 146–166 (1989)

6. Belnap, N.D.: A Useful Four-Valued Logic. In: Modern Uses of Multiple-Valued Logic. D. Reidel (1977)
7. Bruns, G., Huth, M.: Access Control via Belnap Logic: Intuitive, Expressive, and Analyzable Policy Composition. ACM Trans. Inf. Syst. Secur., 1–27 (2011)
8. Crampton, J., Morisset, C.: PTaCL: A Language for Attribute-Based Access Control in Open Systems. In: Degano, P., Guttman, J.D. (eds.) POST 2013. LNCS, vol. 7215, pp. 390–409. Springer, Heidelberg (2012)
9. Ni, Q., Bertino, E., Lobo, J.: D-Algebra for Composing Access Control Policy Decisions. In: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security, ASIACCS 2009, pp. 298–309. ACM (2009)
10. Gurevich, Y., Neeman, I.: DKAL: Distributed-Knowledge Authorization Language. Computer Security Foundations Symposium, 149–162 (2008)
11. Becker, M.Y., Fournet, C., Gordon, A.D.: SecPAL: Design and semantics of a decentralized authorization language. Journal of Computer Security, 619–665 (2010)
12. Li, N., Mitchell, J., Winsborough, W.: Design of a Role-based Trust-management Framework. In: IEEE Symposium on Security and Privacy, pp. 114–130 (2002)
13. Garg, D., Pfenn, F.: Non-Interference in Constructive Authorization Logic. In: Proceedings of the 19th IEEE Workshop on Computer Security Foundations, CSFW 2006, pp. 283–296. IEEE Computer Society, Washington, DC (2006)
14. Abadi, M.: Access Control in a Core Calculus of Dependency. Electronic Notes in Theoretical Computer Science 172, 5–31 (2007)
15. Fitting, M.: Bilattices in Logic Programming. In: Proceedings of the Twentieth International Symposium on Multiple-Valued Logic, pp. 238–246 (1990)
16. Marinovic, S., Craven, R., Ma, J., Dulay, N.: Rumpole: A Flexible Break-glass Access Control Model. In: Symposium on Access Control Models and Technologies, SACMAT 2011, pp. 73–82. ACM (2011)
17. Dong, C., Dulay, N.: Shinren: Non-monotonic Trust Management for Distributed Systems. In: Nishigaki, M., Jøsang, A., Murayama, Y., Marsh, S. (eds.) IFIPTM 2010, vol. 321, pp. 125–140. Springer, Heidelberg (2010)
18. Kolovski, V., Hendler, J., Parsia, B.: Analyzing Web Access Control Policies. In: Proceedings of the 16th International Conference on WWW, pp. 677–686. ACM (2007)
19. Tsankov, P., Marinovic, S., Dashti, M.T., Basin, D.: Decentralized Composite Access Control. Technical report, ETH Zurich (2014), `http://dx.doi.org/10.3929/ethz-a-010045530`
20. Apt, K.R., Blair, H.A., Walker, A.: Towards a Theory of Declarative Knowledge. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 89–148. Morgan Kaufmann Publishers Inc. (1988)
21. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
22. Vardi, M.Y.: The Complexity of Relational Query Languages (Extended Abstract). In: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, STOC 1982, pp. 137–146. ACM, New York (1982)
23. Shmueli, O.: Decidability and Expressiveness Aspects of Logic Queries. In: Proceedings of the ACM Symposium on Principles of Database Systems. ACM (1987)
24. Rissanen, E.: XACML 3.0 Additional Combining Algorithms Profile Version 1.0. Technical report, Axiomatics
25. OASIS: eXtensible Access Control Markup Language, `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html`
26. Seitz, L., Rissanen, E., Sandholm, T., Firozabadi, B.S., Mulmo, O.: Policy Administration Control and Delegation Using XACML and Delegent. In: Proceedings of the International Workshop on Grid Computing, pp. 49–54. IEEE (2005)