Protecting Privacy with a Declarative Policy Framework

Karsten Martiny, Daniel Elenius, Grit Denker
SRI International
333 Ravenswood Ave
Menlo Park, CA 94025
Email: {firstname.lastname}@sri.com

***Abstract*—This article describes a privacy policy framework that can represent and reason about complex privacy policies. By using a Common Data Model together with a formal shareability theory, this framework enables the specification of expressive policies in a concise way without burdening the user with technical details of the underlying formalism. We also build a privacy policy decision engine that implements the framework and that has been deployed as the policy decision point in a novel enterprise privacy prototype system. Our policy decision engine supports two main uses: (1) interfacing with user interfaces for the creation, validation, and management of privacy policies; and (2) interfacing with systems that manage data requests and replies by coordinating privacy policy engine decisions and access to (encrypted) databases using various privacy enhancing technologies.**

I. Introduction

Privacy and private data sharing have emerged as critical issues in the development and use of enterprise information systems and personal information management. While sharing data is essential for enterprises and individuals, mistakenly sharing data with the wrong partner can result in exploitation, and unnecessarily sharing sensitive, private data—even with a trusted partner—can result in serious harm.

New privacy-preserving technologies such as differential privacy [1] and secure computation [2] offer the potential to share data while maintaining its privacy. However, these technologies still need to be supported by additional technology that helps people make decisions about what data to share with whom and using what protections. A privacy framework must accurately represent data sharing (or non-sharing) policies of stakeholders and a privacy policy decision engine must answer requests for data sharing based on the privacy policies in force at the time of the request.

Fully capturing a policy authority's intent in a formal specification usually leads to large sets of complex policies. Thus, the task of creating privacy policies is time consuming and error prone and is inaccessible to users without experience in formal knowledge representations. Our privacy framework solves this problem by specifying policy objects based on an ontology. Together with a general shareability theory, this provides means to specify expressive privacy policies in a concise way and facilitates the development of intuitive user interfaces for the creation of policies, without requiring knowledge about the technical details of the underlying formalism.

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)

We have implemented this framework in a novel enterprise privacy system that is jointly developed by several research teams under Defense Advanced Research Projects Agency's (DARPA's) Brandeis program[1].

Similiar to the architecture of XACML [3], our high-level architecture is separated into Policy Administration, Decision and Enforcement Points. In this paper, we focus on the *Policy Decision Point* (PDP), which decides whether or not to approve data requests. The PDP supports two main uses: (1) interfacing with user interfaces (*Policy Administration Points*) for the creation, validation, and management of privacy policies; and (2) interfacing with systems (*Policy Enforcement Points, PEP*) that manage data requests and replies by coordinating PDP decisions and access to (encrypted) databases using various privacy enhancing technologies. The ontology is shared between Policy Administration, Decision and Enforcement Points. Our framework uses SRI International's (SRI's) Sunflower system, which extends the Flora[2] language and reasoner with features such as a HTTP REST Web server, Java APIs, editing UIs, and features for producing structured explanations of reasoning results in natural language. Sunflower has previously been used for semantically enhanced virtual learning [4], and financial regulatory compliance [5].

A. Related Work

A number of machine-readable privacy policy languages exist to protect access of sensitive information. Most notable in the context of our work are Ponder [6], EPAL [7], Rei [8], KAoS [9], AIR [10], SecPAL [11], and XACML [3]. A common feature of our framework and all of these languages is that they provide some means of privacy protection through role-based access control policies. However, a key difference between our privacy framework and existing languages is the representation of policy objects. Previous work can be broadly categorized into two approaches: XACML, SecPAL

[1]This work was supported by DARPA and SPAWAR under contract N66001-15-C-4069. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright notation thereon. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

[2]http://flora.sourceforge.net/

and Ponder require a unique identification of targeted resources and a requested object needs to exactly match a policy object in order to trigger a policy decision. If several related resources (e.g., class hierarchies) are to be addressed by data sharing policies in these systems, a large number of overlapping policy specifications is required to address all relevant cases. EPAL alleviates this bottleneck to some extend by tagging resources with category labels and allows to express policies over categories, but cannot handle more sophisticated relationships of resources.

KAoS, Rei, and AIR on the other hand are expressive enough to represent richer relationships between targeted resources. They do so by essentially exposing a complete logic language to the policy author, who is left to define the precise semantics of each policy from scratch, including the meaning and effect of any background theory in the context of each policy. This makes the task of specifying intended policies much more challenging and much less accessible to non-experts. Moreover, defining relevant resource relationships on an individual policy level will likely lead to significant overhead in the policy specifications, and thus, making it hard to ensure that specified policies actually reflect the specifier's intent in every case.

Instead, in our approach, policy authors use the domain-specific ontology together with an axiomatic characterization of general background knowledge, and a general, domain-independent shareability theory. This allows the policy engine to generalize to a high degree, with one policy covering many types of requests. In turn, it allows the policy author to write expressive policies in a concise way, capturing their intent without requiring extensive knowledge of the underlying specification formalism.

Another significant difference between our proposed approach and other privacy languages is the treatment of conditions under which data is shared. Virtually all of the above languages evaluate conditions (such as "share only data for persons over the age of 18") before a policy decision is obtained. A disadvantage of this approach is that data already needs to be accessed by the policy decision engine before an access control decision is made, introducing a tight coupling between the policy engine and the database during the decision process. In addition, this approach will likely reject a majority of potential requests because the requested data does not comply with specified policy conditions. However, a requester cannot adjust her requests if she is oblivious of existing policy conditions: for the aforementioned condition on age, a request would be granted if it already included a restriction on age (e.g., a *WHERE* clause in SQL), but—with the exception of AIR, as discussed below—a requester would never know that such a restriction was required. To overcome these problems, our approach does not evaluate conditions prior to deciding on a request, but instead attaches constraints to a potential decision and forwards this to the Policy Enforcement Point. The PEP in turn will enforce these constraints before actually returning results to the requester, for example, by filtering out data for persons under the age of 18. Some constraints may not be supported by the database, but can be evaluated by the PEP (e.g., checking whether an entity is within a certain sensitive area). Other constraints can only be handled by the database (e.g., adding noise to encrypted data in an encrypted database). Either way, any data returned by the PEP will satisfy the constraints returned by the PDP.

With the exception of AIR, our framework is the only approach that is not only able to return decisions for given requests, but can also provide explanations in natural language for its decisions. This feature is especially helpful for deny decisions: if a requester was provided with a reason why a certain request was denied, she has the potential to refine her request in order to receive a positive sharing decision.

Aside from SecPAL, all of the above privacy languages provide some account of obligation to specify usage restrictions of data after it has been released to requesters, e.g., allowing the usage only for a specific purpose. However, since subsequent data usage is beyond control of the framework's PDP and PEP, it is impossible to enforce such usage restrictions, and instead data requesters need to be trusted to comply with specified obligations. Our framework is intended to be used in scenarios where data requesters are not necessarily highly trusted, and thus policy decisions cannot rely on additional obligations. Consequently, the current version of our framework does not address obligations. However, if future applications require an obligation model, this can easily be integrated through an extension of the constraint vocabulary, without any changes to the underlying framework.

II. Overview of the Privacy Policy Framework

In this section, we describe the ontology as an underlying data representation for our approach, followed by a high-level description of our privacy framework, before discussing the detailed technical aspects of this framework in Section III.

A. *Ontology as a Common Data Model*

To define privacy policies for some domain, we use an ontology as a common data model for that domain. The ontology defines a domain-specific vocabulary and background theory that describes the types of information relevant to the domain. The policy framework uses the ontology to formally describe requested data (i.e. database queries) as well as constraints. Furthermore, as we discuss in Section III, axioms in the ontology can be used to infer sharing decisions. The ontology can be defined directly in Flora, or it can be defined in OWL and translated into Flora by Sunflower.

To illustrate this concept, we briefly introduce the Pandemic use-case. This use-case was developed to provide a context for researching, engineering, and demonstrating privacy-preserving technologies. It involves monitoring a disease outbreak in which a number of communities share data about their residents and their residents' disease status, a number of policy authorities implementing data-sharing policies, and a number of different types of users requesting and receiving data relevant to their roles in this use-case. Figure 1 depicts an excerpt of the Pandemic ontology.

The ontology represents different classes of the domain (shown as rectangles in Figure 1), together with properties associated to each class. Properties can be both used to represent

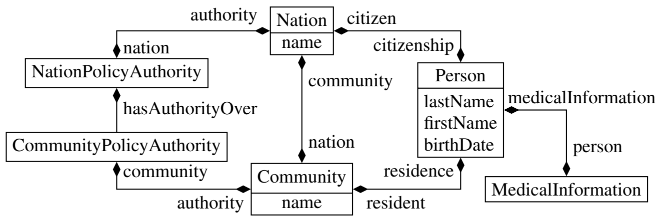

Fig. 1. Pandemic ontology excerpt to illustrate possible links between nations and persons

connections to other classes (depicted by labeled edges), or to represent actual data properties (shown in the lower part of the class representations). As shown in Figure 1, the Pandemic use-case involves nations, communities, policy authorities, and persons. Policy authorities (PAs) can be on a national or on a community level, and there are hierarchical structures between nations and communities, and their corresponding PAs, i.e., communities belong to a nation, and nation PAs have authority over the corresponding community PAs. Each person is associated to a nation and to a community via its `citizenship` and `residence` property, respectively. Moreover, there is medical information associated to each person. In fact, the ontology in the Pandemic use-case has several additional classes associated to the class `MedicalInformation`. However, for the purpose of this article, it suffices to note that there is some representation of medical information. We will return to this ontology excerpt below when we introduce the detailed syntax and semantics of the privacy policy framework and illustrate its use.

B. The Privacy Policy Framework

In Sunflower, we define a generic privacy policy framework and a shareability theory that support the definition of both permissive and restrictive policies.

Each allow or disallow statement is defined with a policy rule. For permissive policies, it is also possible to optionally specify constraints under which a permission is granted. A policy rule has the following components: the policy authority who defines the policy rule, a statement defining it as permissive or restrictive, the data requester(s) to whom the policy rule applies, the type of data that is affected by the policy rule, the time when this policy rule is effective, optionally a set of constraints for permissive policies that define the conditions when sharing is allowed, and finally an identifier and description for the rule.

If a requester wants to access certain data, a request needs to be issued to a policy authority to decide whether—and potentially under what constraints—permission for this request is granted. Every request needs to specify the following parameters,

- The policy authority to be queried
- The data requester
- The requested data
- The time of the request

For such a request, the policy engine checks whether there are corresponding rule definitions that allow or deny the respective request. If there is a matching rule, the policy engine returns a corresponding decision, together with some additional information:

- The decision (allow or don't allow)
- Optionally, the constraints on the allowed data instances
- The identifier and description of the corresponding rule

A single request might trigger different policy rules with conflicting decisions, i.e., one policy rule allows to share the request data, while another policy disallows to share the requested data. In those cases, the Sunflower policy engine must determine the overall desired effect. To achieve this, the policy framework provides a flexible mechanism to designate which policies can override other policies. Our current framework allows for overriding criteria based on priorities, authority hierarchies, and authority delegations. For example, priorities can be used to define baseline policies (such as "don't share anything with anybody") and then create exceptions to this baseline by defining policy rules with higher priorities to allow sharing of specific data.

The privacy policy framework is designed to be used in a *data-independent* PDP, i.e., the policy engine does not have access to actual data instances, but instead derives general sharing decisions. The policy engine is intended to be used in conjunction with one or more standalone database engines, and a PEP that receives the initial data requests. The PEP queries the policy engine to determine if requests are permitted to be translated into actual database queries. Moreover, the PEP needs to ensure that data-specific *constraints* from policy decisions are enforced when processing requests for data (this is discussed in more detail in Section III-E).

III. Syntax and Semantics of the Privacy Policy Framework

In this section, we discuss the aspects of the privacy policy framework in detail and provide precise syntax and semantics of the shareability theory to explain how policy rules can be specified and how they are used to derive sharing decisions for protected data. We explain Flora syntax as it is used, and only to the extent needed to understand the policies. The real policies use fully qualified names with namespace prefixes for all identifiers. In this paper, we omit namespaces for brevity.

Ultimately, we are not planning to expose users to this Flora syntax. Instead, we think of Sunflower/Flora as a *meta-framework* which we use to implement our policy framework. This allows us to quickly experiment with many different language features, while our use cases evolve. Once we feel confident about exactly what language features are needed in real-world policies, we can expose a much simpler language and interface to policy authors.

A. Request Formulas

For most privacy policies, it is not only of interest to control which *values* are allowed to be shared, but also—and in most cases more importantly—to control how different values are allowed to be joined. A well-known study [12] shows that more than 85% of the population of the United States can be uniquely identified by their gender, zip code, and birth date; i.e., if a data set (such as voter registration lists or

patient data from hospitals) contains triples of these attributes, it is possible to identify most persons in the data set. This identification is only possible if the attributes in question are given as triples. On the other hand, three individual sets of genders, zip codes, and birth dates will usually not enable the identification of any individuals, although in both cases exactly the same values are given. The difference is that in the former scenario, connected information between individual values is given, while this information is lacking in the latter scenario. Thus, an effective privacy policy framework clearly requires a specification of possible combinations of individual attributes.

We use a subset of the Flora language itself to specify data requests. A request is described using a Flora formula, which may contain a conjunction of subject-property-object triples, such as `?Community [ resident -> ?Resident ]`, and instance-of formulas, such as `?Resident : Person` (variables are preceded by a question mark, `?`). We call such formulas *Request Formulas (RFs)*. Variables in the formula are free (not quantified), and thus the formula can be interpreted as describing the set of all matching data values.

The classes and properties in the RF must be part of the ontology. Thus, as the ontology provides a common representation of the vocabulary of the domain, the RFs provide a common representation of requests for data in the same domain. At runtime, the requests are translated to concrete data-retrieval operations, depending on the types of data backends, e.g., SQL queries, MongoDB queries, HTTP REST calls, etc. For each such data backend, a translation layer is needed in order to translate to and from the ontology-based representation. In our Pandemic use case, the data was stored in a specialized privacy-preserving database using a subset of SQL for queries.

As an example, consider the ontology depicted in Figure 1 and assume we want to request a list of last names of nations' residents. Note that the ontology does not contain a direct connection between nations and their residents, but instead resident information is provided per community, and communities in turn are associated with nations. In Flora, this RF would be represented as

```
RFp = ${ ?Nation : Nation [ name -> ?NationName,
            community -> ?Community ],
         ?Community : Community [ resident -> ?Resident ],
         ?Resident : Person [ lastName -> ?LastName] }.
```

The `${..}` syntax is Flora's *reified formula* construct, which allows us to treat a formula as a term (or object). The comma `,` denotes conjunction, i.e. logical *and*. Note that only the variables `?LastName` and `?NationName` actually represent data properties of the ontology (and thus are elements that could potentially be shared with a data requester). The other variables are required to unambiguously specify how these two data properties are to be connected.

This small example illustrates some important features of the RF representation. First, note that the ontology excerpt from Figure 1 allows for two different ways of joining nation data with person data to identify (i) a nation's residents (through the intermediate class community), and (ii) a nation's citizens. These two associations represent different concepts and will usually result in different data sets. Thus, it does not suffice to simply specify that names of nations should be joined with persons' last names, but it needs to be explicitly specified *how* (i.e., on which properties) the data types are to be joined. Second, even though the above RF specification traverses through the class `Community`, it does not reveal the association between persons and specific communities, since no data type from the class `Community` is contained in the RF.

B. Reasoning about Data Sharing

The concept of Request Formula is used in the privacy policy framework to specify (i) what data is requested in a specific request, and (ii) what data is affected (allowed or disallowed) by a policy rule. However, it does not suffice to check whether the requested RF *exactly* matches the RF in the policy rules. In most cases, a policy author intents to specify policies that capture a large variety of property combinations. Expecting separate specifications for every relevant combination of properties is infeasable, as it puts a heavy burden on the policy author. Instead, a more flexible relation between requested and allowed RFs needs to be defined.

We introduce a unary higher-order predicate S (for "shareable"), which takes a RF as an argument. This predicate is used to reason about what requests are allowed or disallowed by what policies. For an allow policy, we specify a RF that is allowed by the policy, RF_p. As a first approximation, to check whether a request for RF_r is allowed, we must prove that $\exists \bar{x_p} : S(RF_p) \models \exists \bar{x_r} : S(RF_r)$, where $\bar{x_p}$ is the set of variables in RF_p, and $\bar{x_r}$ is the set of variables in RF_r. Conversely, for a disallow policy to trigger, we must check whether $\exists \bar{x_r} : S(RF_r) \models \exists \bar{x_p} : S(RF_p)$. Here, $\models$ is entailment in the underlying logic (Flora). Roughly, we can think of allow policies as allowing any *subset* of RF_p, and disallow policies as disallowing any *superset* of RF_p. A simple example gives an intuition for why this makes sense: Let's consider again the example from [12], where a triple of gender, zip code, and birth date is used to uniquely identify persons. If such an identification should be prevented, we need an according disallow policy for such triples. Any singletons or pairs in this set usually don't contain enough information to identify individual persons, and thus can be shared without compromising the policy's intend. On the other hand, any supersets (i.e., sets containing this triple and additional data types) clearly contain (more than) enough information to identify individual persons and thus should be denied.

A number of axioms for S allow us to infer what can be shared (or not shared), from what has been explicitly stated. For example if $S((x, y))$ (the conjunction of the formulas x and y) can be shared, then $S(x)$ can be shared. We can also take into account additional background information specified in the ontology. For example, we may have axioms (not shown here) specifying that a property `residence` is the *inverse* of the property `resident`, i.e. if x is a resident of y, then y is the residence of x. We can then incorporate this type of background information through an axiom that says that if `?x[?p->?y]` is shareable, and `?p` is the inverse of `?r`, then `?y[?r->?x]` is shareable. Similarly, given that a superclass can be shared, we allow sharing of all its subclasses. These axioms can be customized for a particular domain, and form

```
@!{NationsAllowDiseaseStatesToRCs}
?pa [ allow_sa(?requester, ?reqData, ?time, ?constr, ?id, ?descr, 0) ] :-
  ?id = "NationsAllowDiseaseStatesToRCs"^^\string,
  ?descr = "Nations share disease states w Response Coordinators"^^\string,
  ?pa : NationPolicyAuthority,
  ?requester : ResponseCoordinator,
  ?polData = ${ ?pa        [ nation -> ?Nation ],
                ?Nation    : Nation [ community -> ?Community, name -> ?NationName ],
                ?Community : Community [ resident -> ?Resident ],
                ?Resident  : Person [ medicalInformation -> ?MedInfo ],
                ?MedInfo   : DiseaseStatus [ state -> ?MedState ] },
  implies_sharing(?polData, ?reqData, ?constr).

@!{CebuCityProhibitsPersonalInformation}
?pa [ \neg allow_sa(?requester, ?reqData, ?time, [], ?id, ?descr,0) ] :-
  ?id = "CebuCityProhibitsPersonalInformation"^^\string,
  ?descr = "Cebu City prohibits sharing of its residents' data."^^\string,
  ?requester : DataRequester,
  ?pa = CebuCityCommunityPA,
  ?polData = ${ ?pa         : CommunityPolicyAuthority [ community -> ?Community ],
                ?Community  : Community [ resident -> ?Resident ],
                ?Resident   : Person},
  implies_sharing(?reqData, ?polData, []).
```

Fig. 2. Policy rules

what we call a *shareability theory*. We have implemented this theory, as well as the underlying Flora entailment checks, in Flora itself.

We must now modify our proof obligation to include the background knowledge B; we must check whether $B \cup \exists \bar{x_p} : S(RF_p) \models \exists \bar{x_r} : S(RF_r)$ (for allow policies), or whether $B \cup \exists \bar{x_r} : S(RF_r) \models \exists \bar{x_p} : S(RF_p)$ (for disallow policies). For instance, using the inverse property axiom mentioned above, `RFp` described in the previous section entails that

```
RFr = ${ ?Person : Person [ residence -> ?Community],
         ?Community : Community }.
```

is shareable. Shareability reasoning with a background theory thus allows a policy to generalize over many different requests. Automated reasoning technology allows us to realize the entailment checks in our policy decision engine.

C. Policy Rules

First, we provide some additional notes on the syntax: A Flora rule has an optional label in `@!{..}`, and a head and a body separated by `:-`. Rules work similarly to Prolog rules; they are Horn clauses where the body logically implies the head, and all variables are implicitly universally quantified on the outside of the rule as a whole. We use the infix operators = (unification), == (syntactic equality), > (greater-than), and \= (disunification). `\neg` is classical negation, and `\+` is Prolog-style negation-as-failure.

The actual policy rules in Flora are specified as *stand-alone* predicates, represented with the suffix `_sa`, as shown in Figure 2. These stand-alone predicates give information how a single policy rule would decide on a given request, without considering the interplay with other rules, which could potentially retract decisions of a given policy due to overrides. The rule head specifies the policy rule with the parameters introduced in Section II-B. Using `NationsAllowDiseaseStatesToRCs` from Figure 2 as an example, it specifies that a policy authority `?pa` allows sharing (denoted by `allow_sa`) of requested data `?reqData` with a requester `?requester` at a certain time `?time`, optionally under certain constraints `?constr`. The remaining three parameters provide meta information about the policy rule. `?id` and `?descr` are typed (string) literals, providing a unique identifier and a description of the policy, respectively. The final argument is a numerical priority value. `allow_sa` is a *boolean* property, lacking the usual arrow `->` and object part of a triple. It is also a *parameterized* property, taking `?requester`, `?reqData`, and so on as parameters (in contrast, RDF triples have only two arguments, the subject and the object).

The rule body specifies under which circumstances the rule's head is true, i.e., when the policy rule is triggered. The first two lines of the rule's body simply instantiate meta information for the rule, namely its identifier and description. The next two lines specify that this rule holds for any national policy authority, i.e., `?pa` needs to be an instance of `NationPolicyAuthority`, and accordingly, the requester needs to be a response coordinator. Next, the data allowed by the policy is specified in `?polData`, as a RF. This RF specifies a connection between the names of nations and the medical data of its residents. As discussed above, a nation's residents are identified by joining nations with their communities and in turn joining communities with their residents. Finally, the last line checks that the requested data is shareable according to this policy, using the `implies_sharing` predicate, which implements the shereability theory discussed in Section III-B. This is a Prolog-style predicate, not using the Flora frame syntax (Flora allows you to mix and match these styles). This check can potentially return constraints on the requested data if the policy contains according specifications. We will discuss the constraint mechanism below; for now it suffices to note that the policy specification does not contain any constraints and thus the variable `?constr` will return an empty list `[]`, i.e., the policy would allow sharing of data without any constraints. Note that the rule's body does not contain any specifications for the parameter `?time`, and thus requests

▼Cebu City Community PA does not allow sharing of personal information with Cebu Nation Response Coordinator | using CebuCityProhibitsPersonalInformation
▼Cebu Nation Response Coordinator is a Data Requester | instance of subclass
Cebu Nation Response Coordinator is a Nation Response Coordinator | fact
▼Every Nation Response Coordinator is a Data Requester | transitivity of subclassing
Every Nation Response Coordinator is a Response Coordinator | fact
Every Response Coordinator is a Data Requester | fact
Shareability of request data implies shareability of policy data | using Reverse_Sharing_Implication_Rule

Fig. 3. Natural language explanation for a deny decision

at arbitrary time points can be accepted. The timing parameter can be instantiated to specific values if policies should be only active for specific time points or time intervals.

Disallowing policies use `\neg allow_sa` instead of `allow_sa`. To illustrate how policies can be specified to deny sharing of data types, consider the policy rule `CebuCityProhibitsPersonalInformation` from Figure 2. This policy is issued by a specific community policy authority, namely for the PA of Cebu City. It denies any access to data of its residents. Note that—opposed to the previous allow policy—this policy does not give specific information about addressed data types, but instead only contains the relation between a community and its residents. However, the shareability check in the last line is reversed for neg allow policies and thus this policy is triggered whenever supersets of the policy data are requested. Consequently, this policy comes to effect whenever some information about Cebu City's residents is requested. Negative sharing decisions are always unconstrained and thus, for negative policies we specify constraints as empty lists `[]`.

To illustrate how Sunflower's natural language explanation capabilities can be used, assume that Cebu Nation's response coordinator requests personal information from residents of Cebu City. This triggers the policy `CebuCityProhibitsPersonalInformation`, and accordingly, the request is rejected. Figure 3 shows the corresponding explanation for this decision.

D. Policy Overrides

As stated before, the policy rules introduced in the previous section only define *stand-alone* decisions that do not take the interplay of different policies into account yet. To determine final sharing decisions based on a set of stand-alone policies, the privacy policy framework provides an overriding mechanism, which determines the final decision based on various overriding criteria. The general overriding rules are `FinalAllow` and `DistributeOverridingAllow`, shown in Figure 4. These rules depend on an auxiliary predicate `overrides` to allow for an easy separation of the universal overriding mechanism and use-case specific overriding criteria, as introduced below. The first rule derives a final allow decision (note that the allow predicate in the rule's head now does not have the `_sa` suffix any longer), if (i) there is a stand-alone policy rule that provides a corresponding allow decision, and (ii) there is no stand-alone disallow rule that denies sharing *and* overrides the allow decision. The second rule ensures that final allow decisions are distributed to all other policy authorities that are overridden by this decision. The actual implementation contains dual rules for overriding neg allow decisions, which are omitted here for brevity. Priorities are omitted from the final decision, as they are only required to potentially determine how different individual policy rules override each other, and are no longer needed if a final decision has been reached.

In the Pandemic use case, we use three different kinds of overrides, which we define in the following. Corresponding Flora representations for these overriding rules are in Figure 4.

a) Priority-based overrides: The first criterion for policy overriding is based on priorities of stand-alone policy rules, as defined in `PriorityOverride`. This rule states that if there are two stand-alone decisions issued by the same policy authority, then the decision with the higher priority overrides the decision with the lower priority. A plain `?`, as seen in this rule, denotes an *anonymous* (don't-care) variable.

b) Authority-based overrides: The second type of override, defined in the `HierarchyOverride` rule, uses the fact that the ontology defines a hierarchy of policy authorities. More specifically, the Pandemic use-case has both national and community policy authorities. Per default, all community policy authorities are subordinate to their respective nation authority. Thus, the nation PAs have authority over their subordinate community authorities and can override their decisions. This rule states that a superior policy authority overrides the decision of subordinate authorities—represented by the property `hasAuthorityOver`—unless the superior PA delegates its authority to the subordinate PA. This brings us to the third type of override:

c) Delegation-based overrides: The pandemic use case provides the option for policy authorities to delegate their decision authority for certain requesters to subordinate PAs, for example because another PA might have more accurate information for corresponding sharing decisions. Such an authority delegation is denoted by the predicate `delegatesAuthority`, and is addressed by a corresponding overriding criterion, defined in the `DelegationOverride` rule. This rule is constructed very similar to the previous, hierarchy-based overriding rule. However, note that next to the positive `delegatesAuthority` predicate, the order of the `hasAuthority` property is reversed now, i.e., a subordinate PA overrides a superior PA if the latter has delegated its authority to the former.

To illustrate the effects of policy overrides, consider again the standalone policies from the Section III-C: the first policy (`NationsAllowDiseaseStatesToRCs`) states that all national policy authorities allow sharing of disease state data, while the second policy (`CebuCityProhibitsPersonalInformation`) states that Cebu City's community policy authority forbids sharing of any personal data. If disease state data is requested

```
@!{FinalAllow}
?pa1 [ allow(?rc, ?requested, ?time, ?constraints, ?id, ?descr) ] :-
  ?pa1 [allow_sa(?rc, ?requested, ?time, ?constraints, ?id, ?descr, ?pr1)],
  \+ ( ?pa2 [ \neg allow_sa(?rc, ?requested, ?, [], ?, ?, ?pr2) ],
    overrides(?pa2, ?pa1, ?pr2, ?pr1, ?rc, ?requested, ?time) ).

@!{DistributeOverridingAllow}
?pa2 [ allow(?requester, ?requested, ?time, ?constr, ?id, ?descr) ] :-
  ?pa1 [ allow(?requester, ?requested, ?time, ?constr, ?id, ?descr) ], ?pa1 \= ?pa2,
  overrides(?pa1, ?pa2, ?, ?, ?requester, ?requested, ?time).

@!{PriorityOverride}
overrides(?pa1, ?pa2, ?pr1, ?pr2, ?, ?, ?) :-
  ?pa1 == ?pa2, ?pr1 : Priority, ?pr2 : Priority, ?pr1 > ?pr2.

@!{HierarchyOverride}
overrides(?pa1, ?pa2, ?, ?, ?requester, ?requested, ?time) :-
  ?pa1 [ hasAuthorityOver -> ?pa2 ], ?pa1 \= ?pa2,
  \+ ?pa1 [ delegatesAuthority(?pa2,?requester, ?requested,?time) ].

@!{DelegationOverride}
overrides(?pa1, ?pa2, ?, ?, ?requester, ?requested, ?time) :-
  ?pa2 [ hasAuthorityOver -> ?pa1 ], ?pa1 \= ?pa2,
  ?pa2 [ delegatesAuthority(?pa1,?requester, ?requested,?time) ].
```

Fig. 4. Override rules

from Cebu City's PA, the combination of these two policies in fact gives rise to a policy override: the latter policy is defined from a community PA's perspective. In the Pandemic use-case, every community PA has a superior nation PA, and since the former policy allows sharing of disease-state data for *all* national PA's, the decision from Cebu City's community PA will be overridden by the superior nation PA; thus, a request for disease state data of Cebu City's residents will be granted, even though Cebu City itself has a rule to deny this request. Note this override is only triggered if the request RF matches the RF defined in the allow policy. Requests for any other personal data are not governed by the allow policy, and thus Cebu City's policy to deny personal data would still be active for such requests.

The overriding approach in our policy framework provides a very flexible way of tailoring overriding criteria to specific needs: The overriding mechanism is defined in a very general way, and specific overriding criteria can be plugged into this mechanism by defining corresponding override rules.

The overriding criteria presented in this section are the ones utilized in the Pandemic use case. It should be noted that this set of overriding criteria does not guarantee a unanimous decision in every possible case. For instance, it could very well happen that a single policy authority defines overlapping allow and disallow policies with the same priorities. Then, some requests might trigger both policy rules. In this case, none of the defined overriding criteria applies, and thus conflicting sharing decisions are returned. In the Pandemic use case, this is the favored behavior because the existence of such a conflicting result can be used to signal conflicts, so that the policy authority becomes aware of these conflicts and can take potential measures to refine the set of policies. If an automatic conflict resolution is favored instead, one could, for example, easily add an additional cautious overriding criterion to ensure that neg allow decisions always take precedence over allow decisions.

E. Constraining Policy Decisions

The policy engine is not only able to return plain allow or deny decisions, but it can also refine allow decisions by optionally attaching constraints to specific allowed data items. For instance, a policy authority might be willing to share personal information of residents over the age of 13 years, while retaining the more sensitive data of young children.

Since the policy engine only derives general sharing decisions without access to actual data, constraints cannot be evaluated within the policy engine. Instead, the constraints are returned, along with the sharing decisions, to the PEP, which must understand the constraint vocabulary. The PEP is then able to enforce those constraints when fetching instance data from data back-ends, before returning results to the requester.

Since a single policy rule can be triggered by a variety of requests, we need a flexible mechanism to bind constraints to specific variables in the RFs. To ensure that constraints are returned in the correct context, they should not be attached to policy rules as a whole, but instead to specific data items specified in a policy rule. To achieve maximum flexibility in the constraint specification, we do not only want to allow for a specification of constraints on requested property values, but we also want to be able specify constraints on other property values, which are potentially not even included in the request.

The policy `NationsAllowConstrainedDiseaseStatesToRCs` shown in Figure 5 illustrates our constraint framework. This policy is a modification of the allow policy (`NationsAllowDiseaseStatesToRCs`) from Section III-C. The additions in this policy specify, using the special `constraints` property, that constraints specified in the variable `?constr` are attached to the *triggering* variable `?Resident`. Whenever a request triggers this variable, the corresponding constraint is returned. A variable v from the policy's data RF_p is triggered if it is required to satisfy the entailment relation between RF_p and the requested data RF_r.

Constraints are always specified such that they contain a

```
@!{NationsAllowConstrainedDiseaseStatesToRCs}
?pa [ allow_sa(?requester, ?reqData, ?time, ?constr, ?id, ?descr, 0) ] :-
  ?id = "NationsAllowConstrainedDiseaseStatesToRCs"^^\string,
  ?descr = "Nations share disease states w Response Coordinators"^^\string,
  ?pa : NationPolicyAuthority,
  ?requester : ResponseCoordinator,
  ?polData = ${ ?pa        [ nation -> ?Nation ],
                ?Nation    : Nation [ community -> ?Community, name -> ?NationName ],
                ?Community : Community [ resident -> ?Resident ],
                ?Resident  : Person [ medicalInformation -> ?MedInfo ],
                ?MedInfo   : DiseaseStatus [ state -> ?MedState ],
                ?Resident [ constraints -> ?constr ] },
                ?thirteenYears \is 13*365*24*60*60, ?time [ subtractTime(?thirteenYears) -> ?latestTime ],
                ?constr = [ ${ ?Resident  : Person [ birthDate -> ?Birthdate ], timeBefore(?Birthdate, ?latestTime) }  ],
  implies_sharing(?polData, ?reqData, ?constr).
```

Fig. 5. Constrained policy rule: previous parts from policy NationsAllowDiseaseStatesToRCs are shown in gray, new additions in black.

path from the triggering variable to the constraint subject, in this case a path from `?Resident` to its property `birthDate`. The actual constraint is then expressed through the predicate `timeBefore`, signaling the property specified in the constraint data path needs to be before a certain time point, thirteen years from the latest time in this example policy. Note that the value of `?constr` is actually implemented as a list of objects. This way, it is possible to specify multiple constraints for a single triggering object.

With this modified policy, a request involving `?Community [ resident -> ?Resident ]` would still return an allow decision, but now with a constraint on the resident's birthday, as shown in `?constr` above. In contrast, a request for, say, only nation names, i.e., `?Nation [name -> ?NationName]` would return an unconstrained allow decision, because the constraint attached to personal data is not triggered by this request.

The path to the constraint subject together with the constraint keyword is then passed on to the PEP, which needs to ensure that these constraints are enforced while retrieving data and before actually returning any data to the requester. Any meaningful constraint can be expressed in the policy framework, but to be enforceable, the PEP has to be aware of the semantics of the constraint vocabulary used.

Examples of defined constraints in the Pandemic use-case include among others temporal constraints (e.g., "time before", "time after"), aggregation constraints (e.g., "only share aggregated counts of a value"), differential privacy constraints (i.e., "only share aggregations with additional noise"), and geographical constraints (e.g., "only share for entities in a specific region").

IV. Conclusions and Future Work

Our approach to capturing and reasoning about privacy policies is the first step in an ambitious project. Using an ontology together with a general shareability theory provides a concise specification of expressive privacy policies. Moreover, using the ontology as a foundation to specify policy objects allows the development of intuitive user interfaces (UI) which make policy creation accessible to users without burdening them with details of the underlying formalism.

Near-term improvements include extending the framework with decision expiration information and more generalized means to detect conflicting policies and providing the user with information about which policies contributed to conflicts. Our longer-term goals are to provide user interfaces for analyzing policies (and their complex interactions), requests, and exposure to privacy risks; and for editing and creating policies that provide the needed protections, while still allowing productive data-sharing to take place. Another direction we wish to explore is to study performance of the policy engine with very large policy bases (in our experiments to date, the policy engine returns answers within a second). Note that the size of the *database* does not matter, because the policy engine is decoupled from the data itself; it reasons only over policy rules and metadata.

References

[1] C. Dwork, "Differential privacy: A survey of results," in *Theory and Applications of Models of Computation: 5th International Conference, TAMC 2008*, pp. 1–19, Springer Verlag, April 2008.

[2] D. W. Archer, D. Bogdanov, B. Pinkas, and P. Pullonen, "Maturity and performance of programmable secure computation," *IEEE Security & Privacy*, vol. 14, no. 5, pp. 48–56, 2016.

[3] OASIS Standard, "eXtensible Access Control Markup Language (XACML) Version 3.0," January 2013. [Online].

[4] D. Elenius, G. Denker, and M. Kim, *Semantically Enhanced Virtual Learning Environments Using Sunflower*, pp. 81–93. Cham: Springer International Publishing, 2016.

[5] R. Ford, G. Denker, D. Elenius, W. Moore, and E. Abi-Lahoud, "Automating financial regulatory compliance using ontology+rules and sunflower," in *Proceedings of the 12th International Conference on Semantic Systems*, SEMANTiCS 2016, (New York, NY, USA), pp. 113–120, ACM, 2016.

[6] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder specification language," in *Policy 01: Workshop on Policies for Distributed Systems and Networks,*, January 2001.

[7] P. Ashley, S. Hafa, G. Karjoth, C. Powers, and M. Schunter, "Enterprise policy authorization language," 2003. [Online].

[8] L. Kagal, T. Finin, and A. Joshi, "A policy language for a pervasive computing environment," in *Policy 03: 4th International Workshop on Policies for Distributed Systems and Networks,*, June 2003.

[9] A. Uszok, J. M. Bradshaw, and R. Jeffers, "KAoS: A policy and domain services framework for grid computing and semantic web services," in *iTrust 2004, Second International Conference on Trust Management*, March 2004.

[10] L. Kagal, C. Hanson, and D. Weitzner, "Using dependency tracking to provide explanations for policy management," in *2008 IEEE Workshop on Policies for Distributed Systems and Networks*, pp. 54–61, June 2008.

[11] M. Becker, C. Fournet, and A. Gordon, "SecPAL: Design and semantics of a decentralized authorization language," *Journal of Computer Security*, vol. 18/4, pp. 619–665, January 2010.

[12] L. Sweeney, "Simple demographics often identify people uniquely," *Carnegie Mellon University, Data Privacy*, 2000.