Expiring Decisions for Stream-based Data Access in a Declarative Privacy Policy Framework

Karsten Martiny
SRI International
333 Ravenswood Ave
Menlo Park, CA 94025
karsten.martiny@sri.com

Grit Denker
SRI International
333 Ravenswood Ave
Menlo Park, CA 94025
grit.denker@sri.com

ABSTRACT

This paper describes how a privacy policy framework can be extended with timing information to not only decide *if* requests for data are allowed at a given point in time, but also to decide for *how long* such permission is granted. Augmenting policy decisions with expiration information eliminates the need to reason about access permissions prior to every individual data access operation. This facilitates the application of privacy policy frameworks to protect multimedia streaming data where repeated re-computations of policy decisions are not a viable option. We show how timing information can be integrated into an existing declarative privacy policy framework. In particular, we discuss how to obtain valid expiration information in the presence of complex sets of policies with potentially interacting policies and varying timing information.

ACM Reference Format:
Karsten Martiny and Grit Denker. 2018. Expiring Decisions for Stream-based Data Access in a Declarative Privacy Policy Framework. In *2nd International Workshop on Multimedia Privacy and Security (MPS '18), October 15, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3267357.3267361

1 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.

Time is important to data sharing decisions in the following two ways: First, in enterprise data sharing settings it is often required to define policies which are not active at all times, but rather policies should be defined with a specific start time, end time, or both. Accordingly, requests for data should then only be affected by a policy if the request time falls within the time window defined by the policy's start and end times. Second, in any scenario involving multimedia streaming data, it is not feasible to compute sharing decisions prior to every individual data access operation because of performance. Thus, privacy decisions should not only decide *if* access to a requested data item is granted, but also for *how long* access is granted, i.e., decisions should be enhanced with expiration information. This would enable a policy framework to allow access the streaming data for specified time windows and automatically revoke access permissions once the expiration time is reached. In the context of large sets of interacting privacy policies, expirations of a single decision must not only consider the deciding policy, but also restrictions imposed by policies that come into effect in the future.

In [11], we introduced a declarative policy framework based on semantic technologies for enterprise privacy systems. This framework makes the specification of privacy policies available to users without experience in formal knowledge representations. The policy author can express concise privacy policies using the vocabulary of the application domain (modeled as ontology) and intuitive interfaces for policy creation [15]. Our framework does not require the user to have knowledge about the technical details of the underlying formalism. And a general shareability theory in our framework allows policy authors to specify with one policy a large variety of property combinations of the domain-specific ontology rather than needing to specify separate policies for every relevant combination of properties.

We have implemented this framework to provide automated policy decisions for data requests and have applied it 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]. Similar to the architecture of XACML [14], our high-level architecture is separated into Policy Administration, Decision and Enforcement Points. The concepts described in this paper are part of the *Policy Decision Point* (PDP), which decides whether or not to approve data requests. Our framework uses SRI International's (SRI's) Sunflower system[2], which extends the Flora[3] 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.

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

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor, or affiliate of the United States government. As such, the United States government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for government purposes only.
MPS '18, October 15, 2018, Toronto, ON, Canada
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5988-7/18/10…$15.00
https://doi.org/10.1145/3267357.3267361

[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://https://sunflower.csl.sri.com/
[3]http://flora.sourceforge.net/

Sunflower has previously been used for semantically enhanced virtual learning [7], and financial regulatory compliance [8].

In this work, we describe how to extend the framework introduced in [11] with structured information about policies' start and end times and to infer decisions with associated expirations. In the presence of multiple policies with varying timing parameters, valid decision expiration information cannot be inferred by only considering an isolated policy. Instead, decisions from all policies need to be considered to ensure that any issued decision expires if a conflicting policy comes into effect in the future. To achieve this, our policy framework works in two stages: For a given request it is first checked which policies govern this request. For every matching policy, a *stand-alone* decision with associated expiration information is inferred. In a second step, a final decision sharing decision is inferred, taking into account timing information from all individual decisions as well as potential overriding information between different policies.

Similar to the architecture of XACML [14], the high-level architecture is separated into Policy Administration, Decision and Enforcement Points. The concepts described in this paper are part of the *Policy Decision Point* (PDP), which decides whether or not to approve data requests. The PDP's policy engine is implemented using the Flora[4] language and reasoner.

The remainder of this paper is structured as follows. The following section gives a brief overview of related work on policy languages, especially focusing on the temporal capabilities of existing languages. A more comprehensive overview of the mentioned privacy policy frameworks comparing several different aspects can be found in [11]. Next, Section 3 summarizes the key concepts of our existing privacy policy framework and especially shows how this can be used to obtain stand-alone policy decisions. Then, in Section 4 we show how timing information can be integrated into this framework to infer final sharing decisions and associated decision expiration information. Moreover, we outline different applications of the expiration information and discuss how this can be used to maintain a server-side decision cache. Finally, the paper concludes with Section 5.

2 RELATED WORK

The work described in this paper proposes an extension to the privacy policy framework introduced in [11]. A variety of other machine-readable privacy policy languages exist to protect access of sensitive information. Most notable approaches for *static* (i.e., not stream-based) access control are Ponder [6], EPAL [2], Rei [9], KAoS [17], AIR [10], SecPAL [3], and XACML [14]. A common feature of all of these approaches is that they provide some means of privacy protection through role-based access control policies. However, a key difference between our privacy framework and other existing languages is the representation of policy objects. Previous work can be broadly categorized into two approaches: XACML, SecPAL 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. Moreover, it allows for hierarchical definitions of policies, where policy decisions may be overridden due to different priorities or authorities, for example.

Of the related languages, only XACML and SecPAL explicitly support "activation time windows", i.e., the specification of certain time intervals for which a policy is supposed to be active. KAoS, Rei, and AIR do not provide an explicit notion of time, but their rich formalisms would allow to integrate a specification of activation time windows. Even if time windows are specified in one of the existing approaches, they only govern permissible request times, and do not provide information about time windows for issued decisions. I.e., sharing decisions are only issued for single points in time and sharing decisions need to be requested prior to *every* individual data access operation, even if repeated requests still fall within one policy's activation time window. In cases where repeated access to the same data is required (i.e., any kind of accessing stream data), this can result in a significant performance bottleneck. When controlling access to multimedia data such as live video streams, it is not even clear how an atomic data access operation would be defined: One extreme would be to define accessing any frame of a video stream as an atomic data access operation—clearly, it would be infeasible to request permission prior to accessing an individual frame of a video stream. The other extreme would be to define access to the entire data stream as an atomic access operation. In this case, a policy authority would not have any temporal control over its access decisions, because any access permission would hold without time limits. This is especially problematic in combination with activation time windows because once permitted, it would allow a data requester to continue accessing a data stream beyond the end time defined in a policy. Thus, access control mechanisms for stream data require to include expiration timing information into issued decisions in order to maintain control over the temporal aspects of access control.

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

A number of approaches exist to manage access control over data streams, e.g., [5], [4], and [12]. These contributions allow for specifications of access control policies with temporal boundaries on the decisions, and thus provide solutions to the aforementioned problem of specifying expiration information of sharing decisions. However, these existing approaches to manage access control for stream based data are subject to similar limitations outlined above, in that they do not enable a concise way to specify generalizable policies and cannot represent policy hierarchies. By extending the policy framework introduced in [11] with decision expiration information, we show in this paper how the power of a declarative framework can be made accessible to applications such as streaming data that demand fast policy decisions. As we will discuss below, this is particularly challenging if hierarchical policy specifications are used, because decisions from individual policies cannot be considered in isolation, but instead need to take other policies' decisions into account to correctly determine overall decision expirations.

A related line of research investigates how access control can be enforced in data streams, e.g., [16], [13], or [1]. These approaches provide cryptographic solutions to protect the data streams itself and ensure that any access to the data is in compliance with defined access control policies. Thus, these contributions fall into the realm of the policy enforcement point and could be combined with our policy framework to ensure that access decisions as determined by our policy decision point are correctly enforced.

3 OVERVIEW OF THE PRIVACY POLICY FRAMEWORK

In this section, we summarize the main aspects of the privacy policy framework as introduced in [11].

3.1 Ontology as a Common Data Model

To define privacy policies for some domain, an ontology is used 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 outline in Section 3.5, 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.

To illustrate this concept, we briefly introduce a Fishery use-case developed within the Brandeis program. It involves sharing collected data about ship locations from the perspective of the United States with members of the Pacific Islands Forum Fisheries Agency (FFA). A small excerpt of the ontology for this use case is shown in Figure 1: the ontology represents different classes of the domain (shown as rectangles in the figure), together with properties associated to each class. Properties can be both used to represent connections to other classes (depicted by labeled edges), and to represent actual data properties (shown in the lower part of the class representations). Moreover, the ontology organizes different classes in a class hierarchy. Figure 1 shows the relevant part of the ontology to represent ship locations: The central piece of information about location information in the fishery ontology is track data, which associates mobile entities with location information. Ships are a subclass of mobile entities with some ship-specific properties, and geographic locations with latitude and longitude properties are a subclass of the general location class.

We will return to this ontology excerpt below when we introduce sample policies to illustrate syntax and semantics of the privacy policy framework.

3.2 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. 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.

3.3 General Sharing Decisions

In general, subject-property-object triples in Flora are represented as `?subject [ ?property -> ?object ]` (variables are preceded by a question mark, ?). This is used to specify sharing decisions on the highest level through statements `?pa [allow(?req,?dd)]` and `?pa [\neg allow(?req,?dd)]` stating that a policy authority `?pa` decides to allow (resp. deny) a request `?req` with details of the decision specified in `?dd`. In this case, `allow` is a *boolean* property of `?pa`, thus omitting the usual arrow `->` and object specification. Moreover, this property is parametrized, taking `?req` and `dd` as parameters. `\neg` is classical negation in Flora.

To represent the request and decision detail objects, functional terms are defined through the rules shown in Figures 2 and 3. A flora rule has an (optional) label specified 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. The comma `,` denotes conjunction, i.e. logical *and*, the semicolon `;` denotes disjunctions, i.e., logical *or*. The infix operator `=` denotes unification, `isnonvar{?v}` is a built-in meta-property test to check whether `?var` is not an unbound variable, and control flow statements are represented through `\if..\then..\else..`

A `Request` has properties to specify the data requester, the requested data (as a request formula to identify elements in the ontology), and the time of the request. The `DecisionDetails` associated with a sharing decision provide additional details of a sharing decision, namely information (id and description) of the policy responsible for the decision, potential constraints, and start and end time of the sharing decision—we will discuss below in detail how the timing information of a sharing decision is obtained. The use of functional terms to represent requests and decision details are an additional extension to the policy framework as introduced in [11]. Previously, all parameters relevant to a policy were encoded directly as top-level parameters of the `allow` predicate. However, that

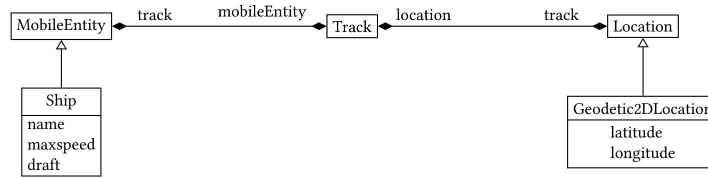

Figure 1: Fishery ontology excerpt to illustrate information about ship locations.

proved to be a rather inflexible approach since adding additional parameters to a policy decision required rewriting of several rules which build on the definition of the `allow` predicate. Now, through the use of functional terms to represent requests and decision details, additional parameters can be added by simply adapting the functional term definitions shown in Figures 2 and 3, while existing rules using specific properties of request or decision details can remain unchanged.

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.

3.4 Request Formulas

A subset of the Flora language itself is used to specify data requests. A request is described using a Flora formula, which may contain a conjunction of subject-property-object triples, such as `?Track [ mobileEntity -> ?Ship ]`, and instance-of formulas, such as `?Ship : Ship`. Such formulas are called *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 the fishery 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 location information for ships together with their respective names. In Flora, this RF would be represented as

```
RFp =  ${ ?Track    : Track
          [ location -> ?Location],
          ?Track
            [ mobileEntity -> ?Ship ],
          ?Location : Geodetic2DLocation
            [ latitude -> ?Latitude ],
          ?Location
            [ longitude -> ?Longitude ],
          ?Ship     : Ship
            [ name -> ?Name ] },
```

The `${..}` syntax is Flora's *reified formula* construct, which allows us to treat a formula as a term (or object). Note that only the variables `?Latitude`, `?Longitude`, and `?Name` 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.

3.5 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 infeasible, as it puts a heavy burden on the policy author. Instead, a more flexible relation between requested and allowed RFs needs to be defined.

As detailed in [11], policies allow to specify affected data using the above concept of request formulas. A formal shareability theory has been developed to characterize the sharing implications of specified policy data wrt. individual requests. A predicate `implies_sharing(?polData, ?reqData, ?constr)` is used to test whether a policy's data specification `?polData` implies that a request for data `?reqData` is permitted, potentially under certain constraints `?constr`. This is a Prolog-style predicate, not using the Flora frame syntax (Flora allows you to mix and match these styles). Roughly speaking, a sharing decision is entailed if the requested data `?reqData` is a subset of the policy's data `?polData`. For negative policies, the sharing implications are reversed, i.e., `implies_sharing(?reqData, ?polData, [])`[5], effectively disallowing requests for all supersets of the data specified in a disallow policy. A set of background knowledge axioms allows to infer additional additional sharing relations from what has been explicitly specified, such as implying *inverse* properties and *subclasses*. Shareability reasoning with a background theory thus allows a

[5]Constraints on shared data make only sense for permissive policies, hence the constraint parameter for negative policies is simply set to the empty list `[]`.

```
@!{PolicyRequestRule}
?pr : Request [ requester -> ?r, requestFormula -> ?rf, requestTime -> ?rt ] :-
   isnonvar{?pr}, ?pr = policyRequest(?r,?rf,?rt).
```

Figure 2: Rule to specify a policy request.

```
@!{DecisionDetailsRule}
?dd : DecisionDetails [ constraints -> ?constr, startTime -> ?st,
expirationTime -> ?et, id -> ?id, description -> ?descr, priority -> ?prio ] :-
   isnonvar{?dd}, ?dd = decisionDetails(?req,?constr,?start,?end,?id,?descr,?prio),
   \if (isnonvar{?start}, ?start [ timeAfter(?req.requestTime) ] )
     \then ?st = ?start \else ?st = ?req.requestTime,
   default_expiration_hours(?hours),
   ?req.requestTime [ addTime(?hours*60*60) -> ?default_exp ] ,
   \if ( isnonvar{?end}, ?end [ timeBefore(?default_exp) ] )
     \then ?et = ?end \else
       ( default_expiration_hours(?hours), ?st [ addTime(?hours*60*60) -> ?et ] ),
   (?et [ timeAfter(?req.requestTime) ]; ?et == ?req.requestTime).
```

Figure 3: Rule to attach details to a sharing decision.

policy to generalize over many different requests. Automated reasoning technology allows us to realize the entailment checks in the policy decision engine. A detailed formal characterization of the shareability theory can be found in [11].

3.6 Policy Rules

The actual policy rules in Flora are specified as *stand-alone* predicates, represented with the suffix _sa, as shown in Figures 4 and 5. These stand-alone allow and disallow 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 3.3.

Using Fisheries_A as an example, it specifies that the policy authority US Navy Organization PA (USNOrganizationPA) allows sharing (denoted by allow_sa) for a given request ?req with additional information about the decision specified in the decision details ?dd. The rule body specifies under which circumstances the rule's head is true, i.e., when the policy rule is triggered. The first rule of the body specifies that data is only shared with requesters whose associated nation is a member of the FFA. Next, the data allowed by the policy is specified in ?polData, as a request formula. This policy uses the RF introduced in Section 3.4 and specifies a connection between the names of ships and their locations via the Track class. Then, it is checked whether the requested data is shareable according to this policy, using the implies_sharing predicate described in Section 3.5. In general, this check will return constraints on the requested data if the policy contains according specifications. For simplicity, we don't use constrained policies in this work, and thus the sharing check will return an empty list [] for this parameter. Next, some meta information about the policy (namely id, description, and priority) is specified and finally, all information about the policy's decision is used to create a DecisionDetails object using the rule DecisionDetailsRule shown in Figure 3. It should be noted that the variables ?startTime and ?endTime are not instantiated in this policy and thus could be replaced by anonymous "don't care" variables, denoted by ? in Flora. We only present this rule with named variables here to help an easier understanding of the meaning of different parameters.

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 Fisheries_E in Figure 5. This policy is used to deny *all* requests for tracks' location property. Note that the data specification of this policy does not explicitly deny access to ship information. 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 into effect for every request that includes tracks' location property. A request purely for ships' names on the other hand would not be affected by this policy. This policy is intended to support a specific 24 hour long mission for which all location information needs to be kept private. Thus, this policy contains specific information about its start and end time, i.e., ?startTime and ?endTime are instantiated to date time values, i.e., spanning the entire day of April 3, 2018. The priority of this policy is set to a higher value (3) than the priority of policy Fisheries_A (1), ensuring that this policy overrides the permissive policy, removing the need to deactivate this permissive policy for the duration of the mission.

The complete use case from the Brandeis project actually specifies several additional policies to model different aspects of the fishery enforcement scenario. We omit those specifications here since policies Fisheries_A and Fisheries_E are sufficient to illustrate the effect of integrating time into policy decisions.

We complete the preliminaries of the policy framework with an example rule for overrides (== denotes syntactic equality in Flora):

```
@!{PriorityOverride}
overrides(?pa1, ?dd1, ?pa2, ?dd2, ?req) :-
  ?pa1 == ?pa2,
  ?dd1 [ priority -> ?pr1 ],
  ?dd2 [ priority -> ?pr2 ],
```

```
@!{Fisheries_A}
USNOrganizationPA [ allow_sa(?req, ?dd) ] :-
   ?req.requester : DataRequester [ associatedNation -> ? [ memberOf -> FFA ]];
   ?polData = ${ ?Track    : Track              [ location -> ?Location],
                 ?Track                         [ mobileEntity -> ?Ship ],
                 ?Location : Geodetic2DLocation  [ latitude -> ?Latitude ],
                 ?Location                       [ longitude -> ?Longitude ],
                 ?Ship     : Ship               [ name -> ?Name ] },
   implies_sharing(?polData, ?req.requestFormula, []),
   ?descr = "US Navy shares ship data with FFA"^^\string,
   ?id = "Fisheries_A"^^\string, ?prio = 1,
   ?dd = decisionDetails(?req, ?constr, ?startTime, ?endTime, ?descr, ?id, ?prio).
```

Figure 4: Fisheries_A policy allows to share ship location information with members of the FFA.

```
@!{Fisheries_E}
USNOrganizationPA [ \neg allow_sa(?req, ?dd) ] :-
   ?req.requester : DataRequester,
   ?startTime = "2018-04-02T00:00:00.000"^^\dateTime,
   ?endTime   = "2018-04-03T00:00:00.000"^^\dateTime,
   ?polData = ${ ?Track : Track [ location -> ?Location] },
   implies_sharing(?req.requestFormula, ?polData, ?constr),
   ?descr = "US Navy imposes blackout of ship location to all others for 24h starting
            midnight Apr, 2"^^\string, ?id = "Fisheries_E"^^\string, ?prio = 3,
   ?dd = decisionDetails(?req, ?constr, ?startTime, ?endTime, ?id, ?descr, ?prio).
```

Figure 5: Fisheries_E denies access for all requesters for a specified time window.

```
  ?pr1 > ?pr2.
```

This rule states that a decision ?dd1 overrides other decisions ?dd2 from the same policy authority, if decision ?dd1 has a higher priority ?pr1. Note that this rule is independent of the actual request ?req (and thus ?req could again be replaced by an anonymous variable ?), but instead applies to all requests for which the corresponding policy authority issues decisions. Overriding rules such as this one illustrate the benefit of specifying policy decisions based on functional terms: if the policy framework will be extended in the future to include additional request or decision detail parameters, only the rules producing the function terms (shown in Figures 2 and 3) need to be adapted (and additional overriding rules using the new parameters could be added), but existing overriding rules such as the shown PriorityOverride continue to work without any change. The policy framework contains additional overriding rules—some dependent on specific requests—such as overriding due to authority hierarchies or authority delegations, but these rules are not relevant for the presented use case.

4 INTEGRATING TIME INTO POLICY SPECIFICATIONS

As shown in the previous section, policy specifications can optionally have absolute values for start time, end time, or both. These parameters can be used to define specific time windows for when a policy is active. Policies without any timing specification are always active.

The effect of timing information of policies on sharing decisions is twofold: First, specified start and end time values specify an "activation time window" to determine whether a policy is triggered by a request in the first place: a request is always issued with the attribute request time, and a policy only applies to a specific request, if the policy does not have (i) a start time which is later than the request time and (ii) an end time which is earlier than the request time. Second, the timing information is used to determine a valid time window for policy decisions, i.e., policy decisions don't apply to an isolated point in time, but instead extend for some time into the future and expire eventually.

These two temporal contributions exhibit some inter-dependencies: Some previous approaches support policy activation time windows by simply checking for an individual policy whether the request time is within the policy's time interval—the approach thus far enabled this by having corresponding checks in the rule's body. However, this approach is insufficient if decisions should also have expiration times in compliance with all defined policies. For instance, any allow decision from our sample policy Fisheries_A issued prior to April 2, 2018 (the start time of policy Fisheries_E) should clearly take the latter policy into account when determining expiration information. If request times were checked against a policy's activation time in isolation, Fisheries_E) would never give information about future decisions, and thus expiration times could not be adapted accordingly. Thus, checking whether requests fall within a policy's activation time window needs to be lifted from individual policies to a higher level.

4.1 Expirations of stand-alone decisions

To determine correct expirations for policy decisions in complex sets of policy, several factors need to be considered: first, it is intuitive to use a policy's end time specification to determine expiration

times of decisions from this policy—any decisions derived from a specific policy should expire at the latest when the policy itself expires. However, policies do not necessarily have an end time parameter. Also, even if an end time for a policy is defined, this time point might be far in the future. In the context of enterprise privacy, it is often desirable to limit validity of decisions from such policies to a shorter period of time. For these situations, a *default expiration interval* can be defined through the predicate `default_expiration_hours`. In the fishery enforcement scenario, the default expiration interval for any decision is set to 24 hours, i.e., `default_expiration_hours(24)`. This is a user-definable predicate and can be set to a value that best fits the application context.

With these considerations, we now return to `DecisionDetailsRule` from Figure 3 to illustrate how timing information is determined for stand-alone policy decisions[6]. This rule defines a time window for each decision as follows:

- If the policy defines a specific start time (checked via the built-in flora predicate `isnonvar{?start}`) that is later than the request time (`?start [timeAfter(?req.requestTime)]`), the decision's start time is set to the policy's start time (i.e., to a future time point), otherwise the decision's start time is set to the time of the request.
- Similarly, for policies with an explicitly defined end time (`isnonvar{?end}`) that is earlier than the default expiration time (`?end [timeBefore(?default_exp)]`), the decision's end time is set to the policy's end time, otherwise is it set to the request time plus the default expiration interval.

4.2 Inferring Final Sharing Decisions

As a result, the policy definitions define a set of *stand-alone* decisions from each policy for a specific request. As already described in [11], a set of overriding rules—such as the priority override `PriorityOverride` shown in Section 3.6—is used to derive a final sharing decision from these stand-alone decisions while observing several overriding criteria. This rule set is now enhanced with additional timing information to derive expirations of final decisions, taking into account potential interactions between different stand-alone decisions.

To illustrate the effect of interacting policies on the expiration times, consider the example policies `Fisheries_A` and `Fisheries_E` together with the defined default expiration interval of 24 hours. Usually, any requests from FFA members for information on ships' locations will be allowed by `Fisheries_A` and the corresponding decisions will be valid for 24 hours from the time of the request. However, there is the temporary blackout policy `Fisheries_E`, which prohibits sharing of ships' locations from midnight, April 2 to midnight, April 3. Thus, any requests for ship locations issued at some time on April 1 (i.e., less than 24 hours before the blackout policy comes into effect) will be allowed, but with a shorter-than-default expiration time, namely the start of the opposing decision from `Fisheries_E`. To illustrate the effect of interacting policies on decision expirations, Table 1 shows resulting decisions with corresponding expiration times for varying request times.

[6]This rule uses some additional helper predicates for time operations. The definitions of these helpers are not shown here for brevity; their functions should be self-explanatory.

This behavior is implemented through the rule `FinalAllow` depicted in Figure 6 and derives a final result in two parts:

First, a final sharing decision `?pa1 [allow(?req,dd)]` is derived from a policy authority `?pa1` for a request `?req` if (i) there is a corresponding stand-alone decision `?pa1 [allow_sa(?req,dd)]`, (ii) the request time falls within the time window defined by the stand-alone decision, and (iii) there is no stand-alone deny decision which overrides the decision from `?pa1` according to overriding criteria.

Second, after a final sharing decision has been found according to the above conditions, the second part of the rule ensures that the expiration time of the decision is set in correspondence with potentially interrupting later rules: if there is an opposing stand-alone decision whose start time falls within the interval of the "surviving" decision, the expiration time is adapted accordingly to comply with the start time of the interrupting decision. Otherwise, the expiration time of the final decision is set to the previously inferred expiration time of the stand-alone decision. This rule shows how a final positive sharing decision is obtained in the absence of overriding negative decisions. A dual rule is used to derive final negative decisions in the absence of overriding positive decisions. For brevity, the dual negative rule is not shown here.

4.3 Usage of decision expiration information

Depending on the available system architecture and specific use case requirements, different usage scenarios of decision expirations are possible. To simplify the following discussion, we assume that data requesters communicate directly with the policy decision point (PDP). In reality, the architecture is centered around a policy enforcement point which mediates between data requesters, policy decision point and actual data stores. We neglect these architectural aspects in the following as they do not impact the general results.

Decision state management on the client side. First, the simplest approach is to consider the decision expiration as some kind of "lease time" and allow the requester to use this result without any restrictions until the expiration time has been reached. This can be implemented for example by issuing access tokens with a corresponding expiration time. The advantage of this approach is the simplicity of its design, and both low implementation and communication overhead. The policy decision point does not need to maintain state information about issued decisions. The requester can store received decisions and can continuously use these decisions to access data (e.g., consuming data streams such as live video) without needing to re-request permission from the policy decision point. This enables real-time access to data streams, since it does not require repeated access requests (which inevitably introduces delays) between the requester and the PDP. The major disadvantage of this approach however is that it is impossible to retract any issued decisions. If a policy authority decides to make changes to the currently defined set of policies, it will take some time (i.e., up to the maximum of the defined default expiration time) until all data access operations are again guaranteed to be in compliance with the updated policy base. Thus, this scheme is not advisable if immediate enforcement of any policy change is required.

Table 1: Examples of final decisions with expiration times for varying request times. The parts of the stand-alone decisions which actually define the final sharing decision are marked in bold face for each request.

request time	stand-alone decisions	final decision
2018-03-31:00:00	Fisheries_A: **allow** (2018-03-31:00:00 – **2018-04-01:00:00**) Fisheries_E: deny (2018-04-02:00:00 – 2018-04-03:00:00)	allow expires: 2018-04-01:00:00 (valid for 24 hours)
2018-04-01:15:00	Fisheries_A: **allow** (2018-04-01:15:00 – 2018-04-02:15:00) Fisheries_E: deny (**2018-04-02:00:00** – 2018-04-03:00:00)	allow expires: 2018-04-02:00:00 (valid for 9 hours)
2018-04-02:10:00	Fisheries_A: allow (2018-04-02:10:00 – 2018-04-03:10:00) Fisheries_E: **deny** (2018-04-02:10:00 – **2018-04-03:00:00**)	deny expires: 2018-04-03:00:00 (valid for 14 hours)

```
@!{FinalAllow}
?pa1 [allow(?req, ?dd)] :-
   ?pa1 [allow_sa(?req, ?dd1)], inTimeWindow(?req,?dd1),
   \naf ( //check if rule currently not overridden -> determines final decision
     ?pa2 [\neg allow_sa(?req, ?dd2)], inTimeWindow(?req,?dd2),
     overrides(?pa2, ?dd2, ?pa1, ?dd1, ?req1) ),
   \if ( //check if rule overridden before expiration -> determines expiration time
     ?minstart = min{?start |
       ?pa3 [\neg allow_sa(?req, ?dd3)], overrides(?pa3, ?dd3, ?pa1, ?dd1, ?req),
        ?start = ?dd3.po#startTime,
        ?start [timeAfter(?req.requestTime), timeBefore(?dd1.expirationTime)] } )
      \then ?dd = resetExpiration(?dd1, ?minstart) \else ?dd = ?dd1.
```

Figure 6: Rule to derive the final sharing decision from stand-alone decisions under consideration of interacting timing information.

Decision state management on the server side. Second, all decision information could be stored purely on the server side. In this scenario, the requester would not need to store expiration information of received decisions at all, but would instead need to poll the PDP prior to any data access operations. The PDP on the other hand would store inferred decisions in a server-side decision cache. Thus, in the case of repeated requests for the same data the PDP would merely need to check whether non-expired decisions already exist in the decision cache and return those decisions without having to invoke the policy reasoner at all to handle the request. Main advantages of this approach are potentially significantly faster response times for repeated requests and—in contrast to the "lease time" approach—the ability to ensure that data access decisions are always in compliance with the current set of policies, without having any delays between changes to the policy base and corresponding sharing decisions. The ability to replace inference tasks with a simple retrieval of cached decisions facilitates the use of the PDP for any kind of scenario, where updated values for the same data elements—such as ship locations—are requested in quick repetitions (e.g., updated ship locations). In the event of policy changes, the server is able to inspect its decision cache and analyze whether stored decisions are impacted by the change. Requests corresponding to the cached decisions can then proactively be re-evaluated by the policy decision engine to prefetch updated decisions. Thus, if a requester repeats a request, it can be served with an already updated decision, even though this decision might have never been issued before.

Within our use case setup, we adopted this approach of using a server-side decision cache to provide fast response times while maintaining the ability to change the set of policies without delays. It should be noted that at first glance it might appear that expiration information is unnecessary if decisions are stored on the server and can be changed at any given time. However, changes to the decision cache need to be triggered by some kind of event. This can easily capture change events to the set of policies. However, relying only on change events to update the decision cache would miss situations where policies are already defined in the policy base, but are not active yet due to a future start time point—such as the previously described blackout policy `Fisheries_E`. Without explicitly determining expiration information of a decision, any positive cached decisions from policy `Fisheries_A` would remain intact in the cache and would be continued to be used to serve future requests, even though the decision should be invalided as soon as

Table 2: Processing times for ship location requests, averaged over 50 executions for each operation.

decision retrieval from	processing time
Non-initialized policy reasoner	7.699 sec
Initialized policy reasoner after change to policy definitions	3.951 sec
Initialized policy reasoner with unchanged policy definitions	0.979 sec
Decision cache	*0.482* sec

the opposing policy `Fisheries_E` comes into effect. On the other hand, in the absence of future opposing policies, the additional information about default expirations would not be required to ensure correct decision result—in these cases, decisions could be cached in the database without any expiration information. The default expiration still serves a practical purpose in these cases in that it provides a simple criterion to remove aged decisions from the cache.

This approach can significantly speed up request processing compared to non-cached decision making and works well in application domains where requests are repeated in quick succession (e.g., accessing updated location data, or frequently refreshing still-images from security cameras). However, even if the request processing itself exhibits a significant speed up, the requirement to send access requests prior to every individual access operation still poses a bottleneck which hinders the application of this approach to be used in scenarios where real time data access is required.

Decision state management on both the server and client side. Third, a combination and extension of the previous two approaches would be possible to realize a push architecture: Both the requester and the PDP could maintain state information about issued decisions. Then, a requester could use locally stored information about received decisions to access data without needing to re-request permission prior to every access operation. The PDP on the other hand would also store decision information about issued decisions. In the event of a policy change which impacts stored decisions, the PDP could re-evaluate the corresponding requests and—if needed—send push notifications with updated decisions to the requester. This approach also ensures that sharing decisions are always in sync with the current set of policies. Compared to using only a server-side decision cache, the advantages of this approach are significantly reduced communication overhead and an even better suitability for accessing streaming data, because it is not required to verify permissions prior to every data access operation. Thus, this approach provides the opportunity to be applied to real time data (such as live video), while still maintaining the option to change access policies without any delays. On the other hand, this approach requires the largest implementation efforts, since decision state information needs to be maintained both on the server and the requester side, and the server additionally needs to maintain information about active requesters and corresponding address information to be able to send push notifications. In this approach, the ability to immediately revoke any issued decisions due to changed policies also requires a continuously available communication channel between PDP and requester.

4.4 Performance Impacts

As described above, expiration information from decisions can be used to maintain a server-side decision cache, which can significantly improve performance for repeated requests. To characterize performance contributions of a decision cache, we need to briefly analyze different possible states of the actual reasoner component of the PDP: In our implementation, the PDP can handle sessions from multiple users, each with its own instance of the actual policy reasoner. Sessions are managed through http cookies and expire after some time of inactivity. If a new session is initiated, a new reasoner instance needs to be created, which—among other things—requires to generate several in-memory tables required for query processing. For an existing session and an unchanged set of policies, repetitions of requests show a large performance increase, because the request can be quickly served through the existing reasoner instance. However, the performance drops throughout an existing session if the set of policies is changed. This is due to required re-computation of the reasoner's in-memory tables to accommodate results from an updated set of rules.

Table 2 shows average processing times for requesting ship locations under these different circumstances. These numbers have been measured in our experimental system setup and should not be used to judge the absolute performance of the PDP—there are several parameters that influence execution times, and especially execution times vary for different policy definitions and different requests. Instead, these numbers are meant to illustrate the benefits of a decision cache on the overall system performance. Unsurprisingly, a request in a newly initialized session takes much longer than all operations—these are the situations were a decision cache is most beneficial because it is not necessary to create a new reasoner instance at all. However, the decision cache also brings significant speed ups in the other scenarios: even in the best case of an initialized reasoner together with an unchanged set of policies, the decision cache can cut the overall processing time roughly in half for our sample request. In the case of policy change events, prefetching of results ensures that repeated requests consistently exhibits a low processing time, even though the actual processing time at the reasoner increases significantly due to the need of updated in-memory tables.

If these technologies were employed in a performance-optimized real-world system, processing times for all of the operations listed in 2 could probably be further reduced, but even the experimental setup clearly shows how the PDP's performance can greatly benefit from caching issued decisions. The request processing time using a decision cache is especially important if decisions are managed purely on the server side—the processing time for an individual request provides a boundary on the frequency of repeated requests.

If the client is involved in the decision state management, i.e., as described in the first and third mode of operation, processing times are not as important, as a request needs to be processed only once prior to access data for a given time window.

5 CONCLUSION AND FUTURE WORK

In this work, we have shown how sharing decisions in a privacy policy framework can be extended to not only decide *if* a request for data is allowed, but also for *how long* a decision is valid. Thus, sharing decisions are no longer issued for single points in time, but instead for time periods. This removes the requirement to invoke the policy reasoner prior to every individual data access operation, which makes privacy policy frameworks available to domains using stream data, while allowing a policy authority to maintain full control over sharing decisions.

REFERENCES

[1] Dinh Tien Tuan Anh and Anwitaman Datta. 2014. Streamforce: Outsourcing Access Control Enforcement for Stream Data to the Clouds. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. ACM, 12.

[2] P. Ashley, S. Hafa, G. Karjoth, C. Powers, and M. Schunter. 2003. Enterprise Policy Authorization Language. https://www.w3.org/2003/p3p-ws/pp/ibm3.html [Online].

[3] Moritz Becker, Cedric Fournet, and Andy Gordon. 2010. SecPAL: Design and Semantics of a Decentralized Authorization Language. *Journal of Computer Security* 18/4, 4 (January 2010), 619–665.

[4] Barbara Carminati, Elena Ferrari, Jianneng Cao, and Kian Lee Tan. 2010. A Framework to Enforce Access Control over Data Streams. *ACM Trans. Inf. Syst. Secur.* 13, 3, Article 28 (July 2010), 31 pages. https://doi.org/10.1145/1805974.1805984

[5] Barbara Carminati, Elena Ferrari, and Kian Lee Tan. 2007. Specifying Access Control Policies on Data Streams. In *Advances in Databases: Concepts, Systems and Applications*. Springer Berlin Heidelberg, Berlin, Heidelberg, 410–421.

[6] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. 2001. The Ponder Specification Language. In *Policy 01: Workshop on Policies for Distributed Systems and Networks,*.

[7] Daniel Elenius, Grit Denker, and Minyoung Kim. 2016. *Semantically Enhanced Virtual Learning Environments Using Sunflower*. Springer International Publishing, Cham, 81–93. https://doi.org/10.1007/978-3-319-49157-8_7

[8] Reginald Ford, Grit Denker, Daniel Elenius, Wesley Moore, and Elie Abi-Lahoud. 2016. Automating Financial Regulatory Compliance Using Ontology+Rules and Sunflower. In *Proceedings of the 12th International Conference on Semantic Systems (SEMANTiCS 2016)*. ACM, New York, NY, USA, 113–120. https://doi.org/10.1145/2993318.2993329

[9] Lalana Kagal, Tim Finin, and Anupam Joshi. 2003. A Policy Language for a Pervasive Computing Environment. In *Policy 03: 4th International Workshop on Policies for Distributed Systems and Networks,*.

[10] L. Kagal, C. Hanson, and D. Weitzner. 2008. Using Dependency Tracking to Provide Explanations for Policy Management. In *2008 IEEE Workshop on Policies for Distributed Systems and Networks*. 54–61.

[11] Karsten Martiny, Daniel Elenius, and Grit Denker. 2018. Protecting Privacy with a Declarative Policy Framework. In *Proceedings of the 12th IEEE International Conference on Semantic Computing (IEEE ICSC 2018)*.

[12] Min Mun, Shuai Hao, Nilesh Mishra, Katie Shilton, Jeff Burke, Deborah Estrin, Mark Hansen, and Ramesh Govindan. 2010. Personal Data Vaults: A Locus of Control for Personal Data Streams. In *Proceedings of the 6th International COnference (Co-NEXT '10)*. ACM, New York, NY, USA, Article 17, 12 pages. https://doi.org/10.1145/1921168.1921191

[13] Rimma V. Nehme, Hyo-Sang Lim, and Elisa Bertino. 2013. FENCE: Continuous Access Control Enforcement in Dynamic Data Stream Environments. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy (CODASPY '13)*. ACM, New York, NY, USA, 243–254. https://doi.org/10.1145/2435349.2435383

[14] OASIS Standard. 2013. eXtensible Access Control Markup Language (XACML) Version 3.0. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html [Online].

[15] Mark St. John, Ron Moore, April Martin, Woodrow Gustophson, Manuela Jaramilla, Grit Denker, Karsten Martiny, and Linda Briesemeister. 2018. Enterprise-Level Private Data Sharing: Framework and User Interface Concepts. In *Poster at Applied Human Factors & Ergonomics Conference (AHFE) 2018*.

[16] Cory Thoma, Adam J. Lee, and Alexandros Labrinidis. 2015. CryptStream: Cryptographic Access Controls for Streaming Data. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY '15)*. ACM, New York, NY, USA, 163–165. https://doi.org/10.1145/2699026.2699134

[17] Andrzej Uszok, Jeffrey M. Bradshaw, and Renia Jeffers. 2004. KAoS: A Policy and Domain Services Framework for Grid Computing and Semantic Web Services. In *iTrust 2004, Second International Conference on Trust Management*.