Aggregation Policies for Tuple Spaces

Linas Kaminskas and Alberto Lluch Lafuente(✉)

DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark
linaskmnsks@gmail.com, albl@dtu.dk

Abstract. Security policies are important for protecting digitalized information, control resource access and maintain secure data storage. This work presents the development of a policy language to transparently incorporate aggregate programming and privacy models for distributed data. We use tuple spaces as a convenient abstraction for storage and coordination. The language has been designed to accommodate well-known models such as k-anonymity and (ε, δ)-differential privacy, as well as to provide generic user-defined policies. The formal semantics of the policy language and its enforcement mechanism is presented in a manner that abstracts away from a specific tuple space coordination language. To showcase our approach, an open-source software library has been developed in the Go programming language and applied to a typical coordination pattern used in aggregate programming applications.

Keywords: Secure coordination · Policy languages · Privacy models
Tuple spaces · Aggregate programming

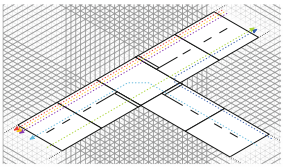
1 Introduction

Privacy is an essential part of society. With increasing digitalization the attack surface of IT-based infrastructures and the possibilities for abuse is growing. It is therefore necessary to include privacy models that can scale with the complexity of those infrastructures and their software components, in order to protect information stored and exchanged, while still ensuring information quality and availability. With EU GDPR regulation [19] being implemented in all EU countries, regulation on how data acquisition processes handle and distribute personal information becomes enforced. This affects software development processes and life cycles as security-by-design choices will need to be incorporated. Legacy systems will also be affected by GDPR compliance. With time, these legacy systems will need to be replaced, not only because of technological advancements, but also due to political and social demands for higher quality infrastructure. No matter the perspective, the importance of privacy-preserving data migration, mining and publication will remain relevant as society advances.

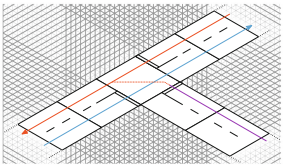
G. Di Marzo Serugendo and M. Loreti (Eds.): COORDINATION 2018, LNCS 10852, pp. 181–199, 2018.
https://doi.org/10.1007/978-3-319-92408-3_8

Aggregation, Privacy and Coordination. Aggregate programming methods are used for providing privacy guarantees (e.g. by reducing the ability to distinguish individual data items), improving performance (e.g. by reducing storage size and communications) and even as the basis of emergent coordination paradigms (e.g. computational field and aggregate programming based on the field calculus [1,23] or the SMuC calculus [14]). Basic aggregation functions (e.g. sums, averages, etc.), do not offer enough privacy guarantees (e.g. against statistical attacks) to support the construction of trustworthy coordination systems. The risk is that less users will be willing to share their data. As a consequence, the quality of different infrastructures and services based on data aggregations may degrade. More powerful privacy protection systems are needed to reassure users and foster their participation with useful data. Fortunately, aggregation-based methods can be enhanced by using well-studied privacy models that allow policy makers to trade between privacy and data utility. We investigate in this work how such methods can be easily integrated in a coordination model such as tuple spaces, that in turn can be used as the basis of aggregation-based systems.

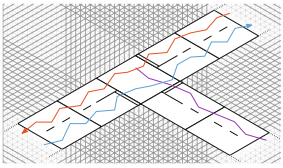
Motivational Examples. One of our main motivations is to address systems where users provide data in order to improve some services that they themselves may use. In such systems it is often the case that: (i) A user decides how much privacy is to be sacrificed when providing data. Data aggregation is performed according to a policy on their device and transmitted to a data collector. (ii) A data collector partitions data by some quality criterion. Aggregation is then performed on each partition and results are stored, while the received data may be discarded. (iii) A process uses the aggregated data, and shares results back to the users in order to provide a service.



(a) Raw data



(b) Aggregated data



(c) Perturbed data

Fig. 1. Different stages of GIS data.

A typical example of such systems are Intelligent Transport System (ITS), which exploit Geographic Information Systems (GIS) data from vehicles to provide better transportation services, e.g. increased green times at intersections, reduction of queue and congestion or exploration of infrastructure quality. As a real world example, bicycle GIS data is exploited by ITS systems to reduce congestion on bicycle paths, while maintaining individuals privacy. Figure 1 shows user positional data in different stages: (a) raw data as collected, (b) data after aggregating multiple trips, and (c) aggregated data with addition of noise to protect privacy. This aggregated data can then be delivered back to the users, in order to support their decision making before more

congestion occurs. Depending on the background knowledge and insights in a service, an adversary can partially or fully undo bare aggregation. By using privacy models and controlling aggregate functions, one can remove sensitive fields such as unique identifiers and device names, and add noise to give approximations of aggregation results. This gives a way to trade data accuracy in favor of privacy.

Another typical example are self-organizing systems. Consider, for instance, the archetypal example of the construction of a distance field, identified in [2] as one of the basic self-organization building blocks for aggregate programming. The typical scenario in such systems is as follows. A number of devices and points-of-interests (PoI) are spread over a geographical area. The main aim of each device is to estimate the distance to the closest PoI. The resulting distributed mapping of devices into distances and possibly the next device on the shortest path, forms a computational field. This provides a basic layer for aggregate programming applications and coordinated systems, as in e.g. providing directions to PoIs. Figure 2 shows an example with the result of 1000 devices in an area with a unique PoI located at $(0, 0)$, where each device is represented by a dot and whose color intensity is proportional to the computed distance. The computation of the field needs to be done in a decentralized way, since the range of communication of devices needs to be kept localized. The algorithm that the devices use to compute the field is based on data aggregations: a device iteratively queries the neighbouring devices for their information and updates its own information to keep track of the closest distance to a PoI. Initially, the distance d_i of each device i is set to the detected distance to the closest PoI, or to ∞ if no device is detected. At each iteration, a device i updates its computed distance d_i as follows. It gets from each neighbour j its distance d_j, and then updates d_i to be the minimum between d_i or d_j plus the distance from i to j. In this algorithm, the key operation performed by the devices is an aggregation of neighbouring data, which may not offer sufficient privacy guarantees. For instance, the exact location of devices or their exact distance to a PoI could be inferred by a malicious agent. A simple case where this could be done is when one device and a PoI are in isolation. A more complex case could be if the devices are allowed to move and change their distances to a PoI gradually. By observing isolated devices and their interactions with neighbours, one could start to infer more about the behaviour of a device group.

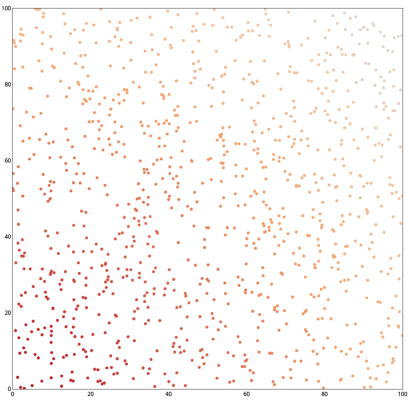


Fig. 2. A distance gradient field (Color figure online)

```
privacy_policy:
    noisy_average:
        aquery avg, data
            altered by result func add_noise
    ...
```

Listing 1.1. An aggregation policy that adds noise to average-based queries

```
program:
    ...
    x := aquery avg, data
    ...
```

Listing 1.2. An aggregated query subject to the policy in Listing 1.1.

Challenges. Engineering of privacy mechanisms and embedding of these directly into large software systems is not a trivial task, and may be error prone. Therefore, it is crucial to separate privacy mechanisms from an application, in such a way that the privacy mechanisms can be altered without having to change the application logic. For example, Listing 1.1 shows a policy that a data hosting service will provide, and Listing 1.1 shows a program willing to use the data. The policy controls the aggregate query (`aquery`) of the program. It only allows to average (`avg`) some `data` and, in addition, it uses a `add_noise` function to the result before an assignment in the program occurs. In this manner, a clear separation of logic can be achieved, and multiple queries of the same kind can be altered by the same policy. Furthermore, it allows policies to be changed at run-time, in order to adapt to changes in regulations or to optimize the implementation of a policy. Separation of concerns provides convenience for both developers and policy makers alike.

Contribution. Our goal is to develop a tool-supported policy language providing access control for which well-studied privacy models, aggregate programming constructs, and coordination primitives could be used to provide non-intrusive data access in distributed applications. We wanted to focus on an interactive setting where data is dynamically produced, consumed and queried, instead of the traditional static data warehousing that privacy models implementations tend to address.

Our first contribution is a novel policy language in Sect. 2 to specify aggregation policies for tuple spaces. The choice of tuple spaces has been motivated by the need to abstract away from concrete data storage models and to address data-oriented coordination models. Our approach to the language provides a clean separation between policies that need to be enforced, and application logic that needs to be executed. The presentation abstracts away from any concrete tuple-based coordination language and we focus on aggregated versions of the traditional operations to add, retrieve and remove tuples.

Our second contribution (Sect. 3) is a detailed description of how two well-studied privacy models such as k-anonymity and (ε, δ)-differential privacy can

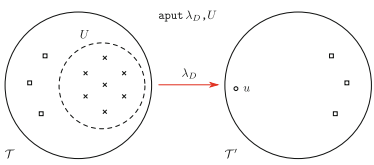
be expressed in our language. For this purpose, those models (which are usually presented in a database setting) have been redefined in the setting of tuple spaces. To the authors knowledge, this is the first time that the definition of those models has been adapted to tuple spaces.

Our third and last contribution (Sect. 4) is an open-source, publicly available implementation of the policy language and its enforcement mechanism in a tuple space library for the Go programming language, illustrated with an archetypal example of a self-organizing pattern used as basic block in aggregate programming approaches [2], namely the above presented computation of a distance gradient.

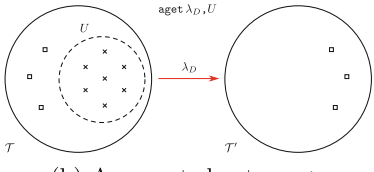
2 A Policy Language for Aggregations

We start the presentation of our policy language by motivating the need of supporting and controlling aggregate programming primitives, and present a set of such primitives. We then move into the description of our policy language, illustrate the language through examples, and conclude the section with formal semantics.

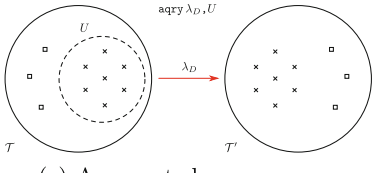
Aggregate Programming Primitives. The main computations we focus in this paper are aggregations of multiset of data items. As we have discussed in Sect. 1, such computations are central to aggregate programming approaches. The main motivation is to control how such aggregations are performed: a data provider could want, for instance, to provide access to the average of a data set, but not to the data set or any of its derived forms. Traditional tuple spaces (e.g. those following the Linda model) do not support aggregations as first-class primitives: a user would need to extract all the data first and perform the aggregation during or after the extraction. Such a solution does not allow to control how aggregations are performed, and the user is in any case given access to an entire set of data items that needs to be protected. However, in databases, aggregate operators can be used in queries, providing thus a first-class primitive to perform aggregated queries, more amenable for access control. A similar situation can be found in aggregate programming languages that provide functions to aggregate data from neighbouring components: the field calculus offers a `nbr`



(a) Aggregated put: `aput`.



(b) Aggregated get: `aget`.



(c) Aggregated query: `aqry`.

Fig. 3. Aggregation primitives.

primitive to retrieve information about neighbouring devices and aggregation is to be performed on top of that, whereas the SMuC calculus is based on atomic aggregation primitives.

We adapt such ideas to tackle the necessity of controlling aggregations in tuple spaces by proposing variants of the classical single-data operations `put/out`, `get/in` and `qry/read`. In particular, we extend them with an additional argument: an aggregation function that is intended to be applied to the multiset of all matched tuples. Typical examples of such functions would be averages, sums, minimum, concatenation, counting functions and so forth. While standard tuple space primitives allow to retrieve *some* or *all* tuples matching some template, the primitives we promote would allow to retrieve the aggregated version of all the matched tuples. More in detail, we introduce the following aggregate programming primitives (Fig. 3):

`aqry`λ_D, U**:** This operation works similarly to an aggregated query in a database and provides an aggregated view of the data. In particular, it returns the result of applying the aggregation function λ_D to all tuples that match the template U.

`aget`λ_D, U**:** This operation is like `aqry`, but removes the matched data with template U.

`aput`λ_D, U**:** This operation is like `aget`, but the result of the aggregation is introduced in the tuple space. It provides a common pattern used to atomically compress data.

It is worth to remark that such operations allow to replicate many of the common operations on tuple spaces. Indeed, the aggregation function could be, for instance, the multiset union (providing all the matched tuples) or a function that provides just one of the matched tuples (according to some deterministic or random function).

Syntax of the Language. The main concepts of the language are knowledge bases in the form of tuple spaces, policies for expressing how operations should be altered, and aggregate programming operators. The language itself can be embedded in any host coordination language, but the primary focus will be in expressing policies. The language is defined in a way that is reminiscent of a concrete syntax for a programming language. Although, the point is not to force a particular syntax, but to have a convenient abstraction for describing the policies themselves. Further, the language and aggregation policies do not force a traditional access control based model by only permitting or denying access to data: policies allow transformations thus giving different views on the same data. This gives a choice to a policy maker to control the accuracy of the information released to a data consumer. Subjects (e.g. users) and contextual (e.g. location) attributes are not part of our language, in order to keep the presentation focused on the key aspects of aggregate programming. Yet, the language could be easily extended to include these attributes.

Table 1. Syntax for policies and aggregate programming operators.

Tuple Space :	$\mathcal{T}$	::=	$\emptyset \mid V : u \mid V : u ; \mathcal{T}$
Policy Label :	v		
Policy Labels :	V	::=	$\emptyset \mid v \mid v , V$
Tuple :	u	::=	$\varepsilon \mid c \mid u_1 , u_2$
Template :	U	::=	$\varepsilon \mid c \mid \Omega \mid U_1 , U_2$
Composable Policy :	Π	::=	$\mathbf{0} \mid \pi \mid \pi ; \Pi$
Aggregation Policy :	π	::=	$v : H$
Aggregation Rule :	H	::=	`none` $\mid$ `put` U `altered by` D_a
			$\mid A_D$ `altered by` $D_U\ D_u\ D_a$
Action :	A	::=	$A_S \mid A_D$
Simple Action :	A_S	::=	`put` $V : u$
Aggregate Action :	A_D	::=	`aput` $\lambda_D , U \mid$ `aget` $\lambda_D , U \mid$ `aqry` λ_D , U
Template Transformation :	D_U	::=	`template func` λ_U
Tuple Transformation :	D_u	::=	`tuple func` λ_u
Result Transformation :	D_a	::=	`result func` λ_a
Aggregate operator :	λ_D	::=	`sum` $\mid$ `avg` $\mid$ `min` $\mid$ `max` $\mid \ldots$
Template operator :	λ_U	::=	`id` $\mid$ `pseudo` $i \mid$ `collapse` $i \mid \ldots$
Tuple operator :	λ_u	::=	`id` $\mid$ `collapse` $i \mid$ `noise` $X \mid \ldots$

The syntax of our aggregation policy language can be found in Table 1. Let Ω denote the types which are exposed by the host language and a type be $\tau \in \Omega$. For the sake of exposition we consider the simple case $\{\texttt{int}, \texttt{float}, \texttt{string}\} \subseteq \Omega$. $\mathcal{T}$ is a knowledge base represented by a tuple space with a multiset interpretation, where the order of tuples is irrelevant and multiple copies of the identical tuples are allowed. For $\mathcal{T}$, the language operator ; denotes the multiset union, $\setminus$ is the multiset difference and $\ominus$ is the multiset symmetric difference. $\mathcal{T}$ contains labelled tuples, i.e. tuples attached a set of labels, with each label identifying a policy. A *tuple* is denoted and generated by u, and an empty tuple is denoted by ε. Tuples may be primed u' or stared $u_\star$ to distinguish between different types of tuples. The type of a tuple u is denoted by $\tau_u = \tau_{u_1} \times \tau_{u_2} \times \ldots \times \tau_{u_n}$. In Sect. 3, individual tuple fields will be needed, and hence we will be more explicit and use $u = (u_1, \ldots, u_i, \ldots, u_n)$, where u_i denotes the i^{th} tuple field with type τ_{u_i}. When dealing with a multiset of tuples of type τ_u (e.g. a tuple space), the type τ_u^* will be used. For a *label set* V, a *labelled tuple* is denoted by $V{:}u$. Similarly as for a tuple, the empty labelled tuple is denoted by ε. A label serves as a unique attribute when performing policy selection based on an *action* A. A template U can contain constants c and types $\tau \in \Omega$ and is used for pattern matching against a $u \in \mathcal{T}$. As with tuples, we shall be explicit with template fields when necessary and use $U = (U_1, \ldots, U_i, \ldots, U_n)$, where U_i denotes the i^{th} template field with type τ_{U_i}. There are three main aggregation actions derived from the classical tuple space operations (`put`, `get`, `qry`), namely: `aput`, `aget` and `aqry`. All operate by applying an aggregate operator λ_D on tuples $u \in \mathcal{T}$ that matches U. Aggregate functions λ_D have a functional type $\lambda_D : \tau_u^* \rightarrow \tau_{u'}$ and

are used to aggregate tuples of type τ_u into a tuple of type $\tau_{u'}$. The *composable policy* Π is a list of policies that contain *aggregation policies* π. An aggregation π is defined by a *policy label* v and an *aggregation rule* H, where v is used as an identifier for H. An aggregation rule H describes how an action A is altered either by a *template transformation* D_U, a *tuple transformation* D_u, and a *result transformation* D_a, or not at all by `none`. A template transformation D_U is defined by a *template operator* $\lambda_U : \tau_U \rightarrow \tau_{U'}$, and can be used for e.g. hiding sensitive attributes or to adapt the template from the public format of tuples to the internal format of tuples. A tuple transformation D_u is defined by a *tuple operator* $\lambda_u : \tau_u \rightarrow \tau_{u'}$. This allows to apply additional functions on a matched tuple u, and can be used e.g. for doing sanitization, addition of noise or approximating values, before performing the aggregate operation λ_D on the matched tuples A *result transformation* D_a is defined by a tuple operator $\lambda_a : \tau_{u'} \rightarrow \tau_{u''}$. The arguments of λ_a are the same as for tuple transformations λ_u, except the transformation is applied on an aggregated tuple. This allows for coarser control, say, in case a transformation on all the matched tuples is computationally expensive or if simpler policies are enough.

Examples and Comparison with a Database. Observe that λ_D and any of the aggregation actions in A can provide all of the aggregate functions found in commercial databases, but with the flexibility of exactly defining how this is performed in the host language itself. The motivation for doing this comes from the fact that: (i) there is tendency for database implementations to provide non-standardized functionalities, introducing software fragmentation when swapping technologies, (ii) user-defined aggregate functions are often defined in a different language from the host language. In our approach, by allowing to directly express both the template for the data needed and aggregate functionality in the host language, helps reducing the programming complexity and improves readability, as the intended aggregation is expressed explicitly and in one place. Moreover, the usage of templates allows to specify the view of data at different granularity levels. For instance, in our motivational example on GIS data, one could be interested in expressing:

1. Field granularity where U contains concrete values only, but access is provided to some fields only. Listing 1.3 shows how to allow access to a specific data source by using $U = (\texttt{"devices"}, \texttt{"gps"}, \texttt{"accelerometer"}, \texttt{"gyroscope"})$ as a template of concrete devices. Here, `id` is the identity function, `first` is an aggregation function which returns the first matched tuple, and `nth 2` selects the second field of the tuple.

```
1 accelerometer-data-only:
2   aqry first, "devices", "gps", "accelerometer", "gyroscope"
3     altered by
4     template func id
5     tuple func id
6     result func (fun x -> nth 2 x)
```

Listing 1.3. Example of field granularity policy.

2. Tuple granularity where U contains concrete values and all fields are provided. Listing 1.4 shows how a policy can provide access to a specific trip. In this case, it is specified by a trip type and trip identifier $U = (\texttt{"bike-ride"}, 1)$.

```
single-ride:
  aqry first, "bike-ride", 1 altered by
    template func id
    tuple func id
    result func id
```

Listing 1.4. Example of tuple granularity policy.

3. Mixed granularity where U contains a mix of concrete values and types. Listing 1.5 shows how this could be used to protect user coordinates expressed as a triplet of `float`'s encoding latitude, longitude, elevation while allowing a certain area. In this case, the `copenhagen` area is exposed, and computation of the average elevation with `avg` is permitted.

```
alices-trips:
  aqry avg, "copenhagen", float, float, float altered by
    template func id
    tuple func (fun x -> nth 4 x)
    result func id
```

Listing 1.5. Example of mixed granularity policy.

4. Tuple-type granularity where U contains only types. Listing 1.6 shows how this could be used to count how many points there are in each discretized part of a map, where `area` maps coordinates into areas.

```
map-partition:
  aqry count, float, float altered by
    template func id
    tuple func (fun (x y) -> area(x,y))
    result func id
```

Listing 1.6. Example of tuple-type granularity policy.

With respect to databases, the aforementioned granularities correspond to: 1. cell level, 2. single row level, 3. multiple row level, and 4. table level Combined with a user-defined aggregate function λ_D and transformations D_U, D_u and D_a, one can provide many different views of a tuple space in a concise manner.

Formal Semantics. Before presenting the formal semantics, we provide a graphical and intuitive presentation using $A = \texttt{aqry}\lambda_D,U$ and some $\mathcal{T}$ and Π as an example shown in Fig. 4. The key idea is: 1. Given some action A, determine the applicable policy π. There can be multiple matches in Π; 2. extract the first-matching policy π with some label v; 3. extract template, tuple and result operators from transformations D_U, D_u, and D_a respectively; 4. extract the aggregate operator λ_D and apply $U' = \lambda_U(U)$; 5. based on the tuples V:u

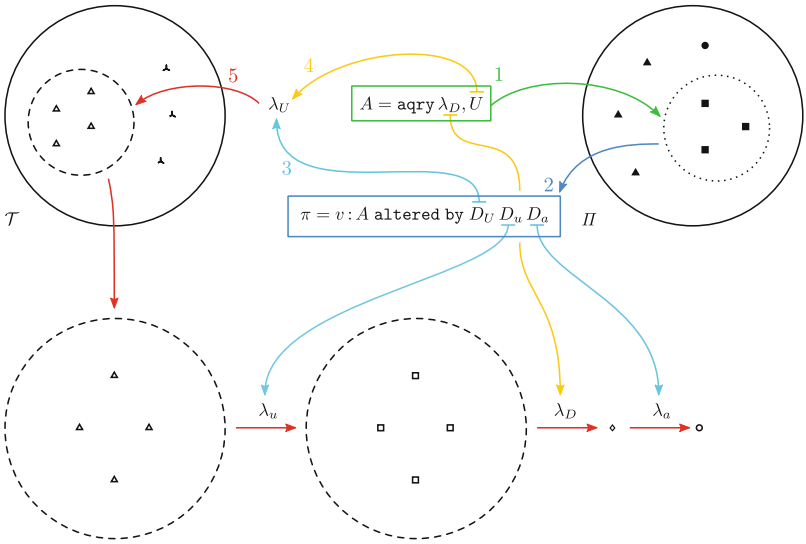


Fig. 4. Semantics of an aggregate action A given an applicable policy π.

from $\mathcal{T}$ that match U' and have $v \in V$: perform tuple transformation with λ_u, aggregation with λ_D, and result transformation with λ_a

The formal operational semantics of our policy enforcement mechanism is described by the set of inference rules in Table 2, whose format is

$$\frac{P_1 \quad \ldots \quad P_i \quad \ldots \quad P_n}{\mathcal{T}, \Pi \vdash A \rightarrow \mathcal{T}', \Pi \rhd R}$$

where $P_1, \ldots, P_n$ are premises, $\mathcal{T}$ is a tuple space subject to Π, A is the action subject to control, and the return value (if any) is modelled by $\rhd R$. The return value may then be consumed by the host language. The absence of a return value denotes that no policy was applicable.

The semantics for applying a policy that matches an aggregate action $\texttt{aput}\lambda_D,U$, $\texttt{aget}\lambda_D,U$ and $\texttt{aqry}\lambda_D,U$ is respectively defined by rules Agg-Put-Apply, Agg-Get-Apply and Agg-Query-Apply. For performing $\texttt{put}V{:}u$, Put-Apply is used. All three rules apply such transformation and differ only in that `aget` and `aput` modify the tuple space. A visual representation of the semantics of $\texttt{aput}\lambda_D,U$, $\texttt{aget}\lambda_D,U$ and $\texttt{aqry}\lambda_D,U$ can be seen in Fig. 4. The premises of the rules include conditions to ensure that the right operation is being captured and a decomposition of how the operation is transformed by the policy. In particular, the set $\mathcal{T}_1$ represents the actually matched tuples (after transforming the template) and $\mathcal{T}_2$ is the actual view of the tuple space being considered (after applying the tuple transformations to $\mathcal{T}_1$). It is on $\mathcal{T}_2$ that the user-defined aggregation λ_D is applied, and then the result transformation λ_a is applied to provide the final result u_a. Rules named Unmatched, Priority-Left, Priority-Left, and Priority-Unavailable take care of scanning the

Table 2. Semantics for action A under Π including the semantics format.

$$\textsc{Put-Apply:}\quad \frac{\Pi = v_\pi : \texttt{put}\ U\ \texttt{altered by result func}\ \lambda_a \quad match(u,U) \quad v_\pi \in V \quad u_a = \lambda_a(u) \quad \mathcal{T}' = \mathcal{T}\,;V:u_a}{\mathcal{T},\Pi \vdash \texttt{put}\ V:u \to \mathcal{T}',\Pi \rhd V:u_a}$$

$$\textsc{Agg-Put-Apply:}\quad \frac{\Pi = v_\pi : \texttt{aput}\ \lambda_D\ U'\ \texttt{altered by template func}\ \lambda_U\ \texttt{tuple func}\ \lambda_u\ \texttt{result func}\ \lambda_a \quad match(U,U') \quad \mathcal{T}_1 = \{V:u \in \mathcal{T} \mid match(u,\lambda_U(U)) \wedge v_\pi \in V\} \quad \mathcal{T}' = \mathcal{T} \setminus \mathcal{T}_1 \quad \mathcal{T}_2 = \{V:\lambda_u(u) \mid V:u \in \mathcal{T}_1\} \quad u_a = \lambda_a(\lambda_D(\{u \mid V:u \in \mathcal{T}_2\})) \quad \mathcal{T}'' = \mathcal{T}'\,;\{v_\pi\}:u_a}{\mathcal{T},\Pi \vdash \texttt{aput}\ \lambda_D\ U \to \mathcal{T}'',\Pi \rhd \{v_\pi\}:u_a}$$

$$\textsc{Agg-Get-Apply:}\quad \frac{\Pi = v_\pi : \texttt{aget}\ \lambda_D\ U'\ \texttt{altered by template func}\ \lambda_U\ \texttt{tuple func}\ \lambda_u\ \texttt{result func}\ \lambda_a \quad match(U,U') \quad \mathcal{T}_1 = \{V:u \in \mathcal{T} \mid match(u,\lambda_U(U)) \wedge v_\pi \in V\} \quad \mathcal{T}' = \mathcal{T} \setminus \mathcal{T}_1 \quad \mathcal{T}_2 = \{V:\lambda_u(u) \mid V:u \in \mathcal{T}_1\} \quad u_a = \lambda_a(\lambda_D(\{u \mid V:u \in \mathcal{T}_2\}))}{\mathcal{T},\Pi \vdash \texttt{aget}\ \lambda_D\ U \to \mathcal{T}',\Pi \rhd \{v_\pi\}:u_a}$$

$$\textsc{Agg-Query-Apply:}\quad \frac{\Pi = v_\pi : \texttt{aqry}\ \lambda_D\ U'\ \texttt{altered by template func}\ \lambda_U\ \texttt{tuple func}\ \lambda_u\ \texttt{result func}\ \lambda_a \quad match(U,U') \quad \mathcal{T}_1 = \{V:u \in \mathcal{T} \mid match(u,\lambda_U(U)) \wedge v_\pi \in V\} \quad \mathcal{T}_2 = \{V:\lambda_u(u) \mid V:u \in \mathcal{T}_1\} \quad u_a = \lambda_a(\lambda_D(\{u \mid V:u \in \mathcal{T}_2\}))}{\mathcal{T},\Pi \vdash \texttt{aqry}\ \lambda_D\ U \to \mathcal{T},\Pi \rhd \{v_\pi\}:u_a}$$

$$\textsc{Unmatched:}\quad \frac{\Pi = v_\pi : \texttt{none} \vee (\Pi = v_\pi : A_2\ \texttt{altered by}\ D_U\ D_u\ D_a \wedge A_1 \neq A_2) \vee \Pi = \mathbf{0}}{\mathcal{T},\Pi \vdash A_1 \to \mathcal{T},\Pi}$$

$$\textsc{Priority-Right:}\quad \frac{\mathcal{T},\Pi_1 \vdash A \to \mathcal{T},\Pi_1 \quad \mathcal{T},\Pi_2 \vdash A \to \mathcal{T}',\Pi_2' \rhd V:u}{\mathcal{T},\Pi_1\,;\Pi_2 \vdash A \to \mathcal{T}',\Pi_1\,;\Pi_2' \rhd V:u}$$

$$\textsc{Priority-Left:}\quad \frac{\mathcal{T},\Pi_1 \vdash A \to \mathcal{T}',\Pi_1' \rhd V:u}{\mathcal{T},\Pi_1\,;\Pi_2 \vdash A \to \mathcal{T}',\Pi_1'\,;\Pi_2 \rhd V:u}$$

$$\textsc{Priority-Unavailable:}\quad \frac{\mathcal{T},\Pi_1 \vdash A \to \mathcal{T},\Pi_1 \quad \mathcal{T},\Pi_2 \vdash A \to \mathcal{T},\Pi_2}{\mathcal{T},\Pi_1\,;\Pi_2 \vdash A \to \mathcal{T},\Pi_1\,;\Pi_2}$$

policy as list. It is up to the embedding in an actual host language to decide what to do with the results. For example, in our implementation, if the policy enforcement yields no result, the action is simply ignored.

3 Privacy Models

The design of our language has been driven by inspecting a variety of privacy models, first and foremost k-anonymity and (ε, δ)-differential privacy. We show in this section how those models can be adopted in our approach. The original definitions have been adapted from databases to our tuple space setting.

k-anonymity. The essential idea of k-anonymity [15,20,22] is to provide anonymity guarantees beyond hiding sensitive fields by ensuring that, when

information on a data set is released, every individual data item is indistinguishable from at least $k-1$ other data items. In our motivational examples, for instance, this could be helpful to protect the correlation between devices and their distances from an attacker that can observe the position and number of devices in a zone and can obtain the list of distances within a zone through a query. k-anonymity is often defined for tables in a database, here it shall be adapted to templates U instead. We start by defining k-anonymity as a property of $\mathcal{T}$: roughly, k-anonymity requires that every tuple u cannot be distinguished from at least $k-1$ other tuples. Distinguishability of tuples is captured by an equivalence relation $=_t$. Note that $=_t$ is not necessarily as strict as tuple equality: two tuples u and u' may be different but equivalent, in the sense that they can be related exactly by the same, and possibly external, data. In our setting, k-anonymity is formalized as follows.

Definition 1 (k-anonymity). *Let $k \in \mathbb{N}^+$, $\mathcal{T}$ be a multiset of tuples, and let $=_t$ be an equivalence relation on tuples. $\mathcal{T}$ has k-anonymity for $=_t$ if:*

$$\forall u \in \mathcal{T}.|\{u' \in \mathcal{T}_U \mid u' =_t u\}| \geq k$$

In other words, the size of the non-empty equivalence classes induced by $=_t$ is at least k. We say that a multiset of tuples $\mathcal{T}$ has k-anonymity if $\mathcal{T}$ has k-anonymity for $=_t$ being tuple equality (the finest equivalence relation on tuples). k-anonymity is not expected to be a property of the tuple space itself, but of the release of data provided by the operations `aqry`, `aget` and `aput`. In particular, we say that k-anonymity is provided by a policy Π on a tuple space $\mathcal{T}$ when for every query based on the above operations the released result u_a (cf. Fig. 4) has k-anonymity. Note that this does only make sense if the result u_a is a multiset of tuples, which could be the case when the aggregation function is a multiset operation like multiset union. Policies can be used to enforce k-anonymity on specific queries. Consider for instance the previously mentioned example of the attacker trying to infer information about distances and positions of devices. Assume the device information is stored in tuples (x, y, i, j, d) where (x, y) are actual coordinates of the devices, (i, j) represents the zone in the grid and d is the computed distance to the closest PoI. Suppose further that we want to provide access to a projection of those tuples by hiding the actual positions and providing zone and distance information. Hiding the positions is not enough and we want to provide 2-anonymity on the result. We can do so with the following policy:

```
2-anonymity:
    aqry mset_union, float, float, int, int, float altered by
       tuple func (fun x y i j d -> if anonymity(2) (i j d) else nil))
```

Listing 1.7. Example of k-anonymity for $k = 2$.

where `anonymity(k)` checks k-anonymity on the provided view $\mathcal{T}_2$ (cf. Fig. 4), according to Definition 1. Basically, the enforcement of the policy will ensure that we provide the expected result, if in each zone there are at least two devices with the same computed distance, otherwise the query produces the empty set.

(ε, δ)-differential Privacy. Differential privacy techniques [7] aim at protecting against attackers that can perform repeated queries with the intention of inferring information about the presence and/or contribution of single data item in a data set. The main idea is to add controlled noise to the results of queries so to reduce the amount of information that such attackers would be able to obtain. Data accuracy is hence sacrificed for the sake of privacy. For instance, in the motivational example of the distance gradient, differential privacy can be used to the approximate the result of the aggregations performed by the gradient computation. This is done in order to minimize leakage about the actual positions and distance of each neighbouring device. Differential privacy is a property of a randomized algorithm, where the data set is used to give enough state information in order to increase indistinguishably. Randomization arises from privacy protection mechanisms based on e.g. sampling and adding randomly distributed noise. The property requires that performing a query for all possible neighbouring subsets of some data set, the addition (or removal) of a single data item produces almost indistinguishable results. Differential privacy is often presented in terms of histogram representations of databases not suitable for our purpose. We present in the following a reformulation of differential privacy for our setting. Let $\mathbf{P}[\mathcal{A}(\mathcal{T}) \in S]$ denote the probability that the output $\mathcal{A}(\mathcal{T})$ of a randomized algorithm $\mathcal{A}$ is in S when applied to $\mathcal{T}$, where $S \subseteq \mathbf{R}(\mathcal{A})$ and $\mathbf{R}(\mathcal{A})$ is the codomain of $\mathcal{A}$. In our setting $\mathcal{A}$ should be seen as the execution of an aggregated query, and that randomization arises from random noise addition. (ε, δ)-differential privacy in our setting is then defined as the following property.

Definition 2 ((ε, δ)-differential privacy). *Let $\mathcal{A}$ be a randomized algorithm, $\mathcal{T}$ be a tuple space, e be Euler's number, and ε and δ be real numbers. $\mathcal{A}$ satisfies (ε, δ)-differential privacy if and only if for any two tuple spaces $\mathcal{T}_1 \subseteq \mathcal{T}$ and $\mathcal{T}_2 \subseteq \mathcal{T}$ such that $\|\mathcal{T}_1 \ominus \mathcal{T}_2\|_{\tau_u} \leq 1$, and for any $S \subseteq \mathbf{R}(\mathcal{A})$, the following holds:*

$$\mathbf{P}[\mathcal{A}(\mathcal{T}_1) \in S] \leq e^{\varepsilon} \cdot \mathbf{P}[\mathcal{A}(\mathcal{T}_2) \in S] + \delta$$

Differential privacy can be enforced by policies that add a sufficient amount of random noise to the result of the queries. There are several noise addition algorithms that guarantee differential privacy. A common approach is based on the *global sensitivity* of data set for an operation and a *differentially private mechanism* which uses the global sensitivity to add the noise. Global sensitivity measures the largest possible distance between neighbouring subsets (i.e. differing in exactly one tuple) of a tuple space, given an operation. The differentially private mechanism uses this measure to distort the result when the operation is applied. To define a notion of sensitivity in our setting, assume that for every basic type τ there is a *norm* function $\|\cdot\| : \tau_u \to \mathbb{R}$ which maps every tuple into a real number. This is needed in order to define a notion of difference between tuples. We are now ready to define a notion of sensitivity for a given aggregate operator λ_D.

Definition 3 (Sensitivity). *Let $\mathcal{T}$ be a tuple space, $\lambda_D : \tau_{u'} \to \tau_{u_\star}$ be an aggregation function, and $p \in \mathbb{N}^+$. The p^{th}-*global sensitivity *Δ_p of λ_D is defined as:*

$$\Delta_p(\lambda_D) = \max_{\substack{\forall\, \mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{T} \\ \mathcal{T}_1 \ominus \mathcal{T}_2 = 1}} \sqrt[p]{\sum_{i \in \{0,..,|\tau_{u'}|\}} |\|\lambda_D(\mathcal{T}_1)_i\| - \|\lambda_D(\mathcal{T}_2)_i\||^p} \tag{1}$$

Roughly, Eq. (1) is expressing that the sensitivity scale of an aggregate operator is determined by the largest value differences between all fields of the aggregated tuples. The global sensitivity can then be used to introduce Laplace noise according to the well-known *Laplace mechanism*, which provides $(\varepsilon, 0)$-differential privacy.

Definition 4 (Laplace noise addition). *Let $\mathcal{T}$ be a tuple space, $\lambda_D : \tau_u^* \to \tau_{u'}$ be an aggregation function, $\oplus : \tau_{u'} \times \tau_{u'} \to \tau_{u'}$ be an addition operator for type $\tau_{u'}$, $\varepsilon \in]0, 1]$, $p \in \mathbb{N}^+$, and $\boldsymbol{Y} = (Y_1, \ldots, Y_i, \ldots, Y_n)$ be a tuple of random variables that are independently and identically distributed according to the Laplace distribution $Y_i \sim \mathcal{L}(0, \Delta_p(\lambda_D)/\varepsilon)$. The* Laplace noise addition *function $\texttt{laplace}_{\mathcal{T},\lambda_D,\varepsilon}$ is defined by:*

$$\texttt{laplace}_{\mathcal{T},\lambda_D,\varepsilon}(u) = u \oplus \boldsymbol{Y} \tag{2}$$

Note that the function is parametric with respect to the noise addition operator $\oplus$. For numerical values $\oplus$ is just ordinary addition. In general, for $\oplus$ to be to be meaningful, one has to define it for any type. For complex types such as strings, structures or objects this is not trivial, and either one has to have a well-defined $\oplus$ or other mechanisms should be considered for complex data types.

Consider again our motivational example of distance gradient computation, we can define a policy to provide differential privacy on the aggregated queries of each round of the computation as follows:

```
1 edp:
2   aqry minD, float, float, int, int, float altered by
3     result func (fun x y i j d  -> (laplace minD 0.9 (x, y, d)))
```

Listing 1.8. Example of $(\varepsilon, 0)$-differential privacy policy.

The policy controls queries aiming at retrieving the information (x, y) coordinates, (i, j) zone and distance d of the device that is closest to a PoI, obtained by the aggregation function `minD`. The query returns only the coordinates of such device and its distance, after distorting them with Laplace noise by function `laplace`, implemented according to Definition 4 (with the tuple space being the provided view $\mathcal{T}_2$, cf. Fig. 4).

More in general, the enforcement of policies of the form

```
1   aquery λD, U altered by
2     template func DU
3     tuple func Du
4     result func (fun u -> (laplace λD ε u))
```

Listing 1.9. Schema for $(\varepsilon, 0)$-differential privacy policies.

provides $(\varepsilon, 0)$-differential privacy on the view of the tuple space (cf. $\mathcal{T}_2$ in Fig. 4) against aggregated queries based on the aggregation function λ_D.

4 Aggregation Policies at Work

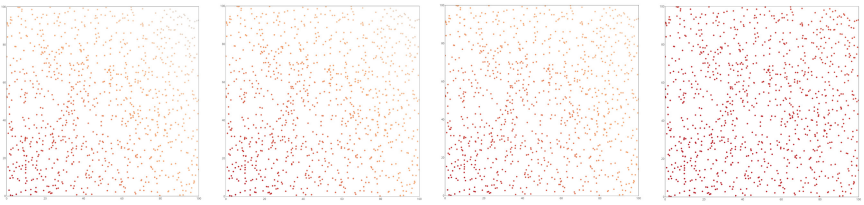
To showcase the applicability of our approach to aggregate computing applications, we describe in this section a proof-of-concept implementation of our policy language and its enforcement mechanism in a tuple space library (cf. Sect. 4.1), and the implementation of one of the archetypal self-organizing building blocks used in aggregate programming, namely the computation of a distance gradient field, that we use also to benchmark the library (cf. Sect. 4.2).

4.1 Implementation of a Proof-of-Concept Library

The open-source library we have implemented is available for download, installation and usage at https://github.com/pSpaces/goSpace. The main criteria for choosing Go was that it provides a reasonable balance between language features and minimalism needed for a working prototype. Features that were considered important included concurrent processes, a flexible reflection system and a concise standard library. The goSpace project was chosen because it provided a basic tuple space implementation, and had the fundamental features, such as addition, retrieval and querying of tuples based on templates, and it also provides derived features such as retrieval and querying of multiple tuples. Yet, goSpace itself was modified in order to provide additional features needed for realizing the policy mechanism. One of the key features of the implementation is a form of code mobility that allows to transfer functions across different tuple spaces. This was necessary to serve as a foundation for allowing user-defined aggregate functions across multiple tuple spaces. Further, the library was implemented to be slightly more generic than what is given in Sect. 2 and can in principle be applied to other data structures beyond tuple spaces and aggregation operators on tuple space. Currently, our goSpace implementation supports policies for the actions `aput`, `aget` and `aqry` but it can be easily extended to support additional operations.

4.2 Protecting Privacy in a Distance Gradient

We have implemented the case study of the distance field introduced in Sect. 1 as a motivational example. In our implementation, the area where devices and PoIs are placed, is discretized as a grid of zones; each device and PoI has a position and is hence located in a zone. The neighbouring relation is given by the zones: two devices are neighbours if their zones are incident. Devices can only detect PoIs in their own zone and devices cannot communicate directly with each other: they use a tuple space to share their information. Each device publishes in the tuple space their information (position, zone and computed distance) labelled with a privacy policy. The aggregation performed in each round uses the `aqry`



(a) noise 0.01 (b) noise 0.1 (c) noise 0.2 (d) noise 1.0

Fig. 5. Distance gradients with aggregation policies based on noise addition. (Color figure online)

operation with an aggregation function that selects the tuple with the smallest distance to a PoI.

Different policies can be considered. The identity policy would simply correspond to the typical computation of the field as seen in the literature. Basically, all devices and the tuple space are considered to be trustworthy and no privacy guarantees are provided. Another possibility would be to consider that devices, and other agents that want to exploit the field, cannot be fully trusted with respect to privacy issues. A way to address this situation would be to consider policies that hide or distort the result of the aggregated queries used in each round.

We have performed several experiments with our case study and we have observed, as expected, that such polices may affect accuracy (due to noise addition) and performance (due to the overhead of the policy enforcement mechanism). Some results are depicted in Figs. 2, 5 and 6. In particular the figures show experiments for a scenario with 1000 devices and a discretization of the map into a 100×100 grid. Figure 2 shows the result where data is not protected but is provided as-it-is, while Fig. 5 shows results that differ in the amount of noise added to the distance obtained from the aggregated queries. This is regulated by a parameter x so that the noise added is drawn from a uniform distribution in $[-x * d, x * d]$, where d is the diameter of each cell of the grid (actually $\sqrt{10 * 10}$). In the figures, each dot represents a device and the color intensity is proportional to the distance to the PoI, which is placed at $(0, 0)$. Highest intensity corresponds to distance 0, while lowest intensity corresponds to the diameter of the area ($\sqrt{200}$). The results with more noise (Fig. 5(d)) make it evident how noise can affect data accuracy: the actual distance seems to be the same for all nodes. However, Fig. 5, which shows the same data but where the color intensity goes from 0 to the

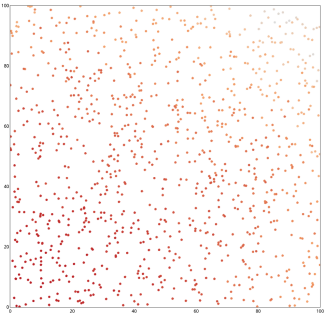


Fig. 6. Gradient with noise

maximum value in the field, reveals that the price paid for providing more privacy does not affect much the field: the gradient towards the PoI is still recognizable.

5 Conclusion

We have designed and implemented a policy language which allows to succinctly express and enforce well-understood privacy models in the syntactic category such as k-anonymity, and in the semantic category such as (ε, δ)-differential privacy. Aggregate operations and templates defined for a tuple space were used to give a useful abstraction for aggregate programming. Even if not shown here, our language allows to express additional syntactic privacy models such as ℓ-diversity [15], t-closeness [13,21] and δ-presence [8]. Our language does not only allow to adopt the above mentioned privacy models but it is flexible enough to specify and implement additional user-defined policies. The policy language and its enforcement mechanism have been implemented in a publicly available tuple space library. The language presented here has been designed with minimality in mind, and with a focus on the key aspects related to aggregation and privacy protection. Several aspects of the language can be extended, including richer operations to compose policies and label tuples, user-dependent and context-aware policies, tuple space localities and polyadic operations (e.g. to aggregate data from different sources as usual in aggregate computing paradigms). We believe that approaches like ours are fundamental to increase the security and trustworthiness of distributed and coordinated systems.

Related Work. We have been inspired by previous works that enriched tuple space languages with access control mechanism, in particular SCEL [6,16] and KLAIM [3,11]. We have also considered database implementations with access control mechanisms amenable for the adoption of privacy models. For example, [5] discusses the development strategies for FBAC (fine-grained access control) frameworks in NoSQL databases and showcases applications for MongoDB, the Qapla policy framework [18] provides a way to manipulate DBMS queries at different levels of data granularity and allows for transformations and query rewriting similar to ours, and PINQ [17] uses LINQ, an SQL-like querying syntax, to express queries that can apply differential privacy embedded in C#. With respect to databases our approach provides a different granularity to control operations, for instance our language allows to easily define template-dependent policies. Our focus on aggregate programming has been also highly motivated by the emergence of aggregate programming and its application to domains of increasing interest such as the IoT [1]. As far as we know, security aspects of aggregate programming are considered only in [4] where the authors propose to enrich aggregate programming approaches with trust and reputation systems to mitigate the effect of malicious data providers. Those considerations are related to data integrity and not to privacy. Another closely related work is [9] where the authors present an extension to a tuple space system with privacy properties based on cryptography. The main difference with respect to our work is in the different privacy models and guarantees considered.

Future Work. One of the main challenges of current and future privacy protection systems for distributed systems, such as the one we have presented here, is their computational expensiveness. We plan to carry out a thorough performance evaluation of our library. We plan in particular to experiment with respect to different policies and actions. It is well known that privacy protection mechanism may be expensive and finding a right trade-off is often application-dependent. Part of the overhead in our library is due to the preliminary status of our implementation where certain design aspects have been done in a naive manner to prioritize rapid prototyping over performance optimizations, e.g. use of strong cryptographic hashing, use of standard library concurrent maps and redundancies in some data structures. This makes room for improvement and we expect that the performance of our policy enforcement mechanism will be significantly improved. More in general, finding the optimal k-anonymity is an NP-hard problem. There is however room for improvements. For instance, [12] provides an approximation algorithm. This algorithm could be adapted if enforcement is needed. An online differentially private algorithm, namely *private multiplicative weights algorithm*, is given in [7]. Online algorithms are worth of investigation since interactions with $\mathcal{T}$ are inherently online. Treatment of functions and functional data in differential privacy setting can be found [10]. We are currently investigating online efficient algorithms to improve the performance of our library.

References

1. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the Internet of Things. Computer **48**(9), 22–30 (2015)
2. Beal, J., Viroli, M.: Building blocks for aggregate programming of self-organising applications. In: Eighth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW 2014, London, United Kingdom, 8–12 September 2014, pp. 8–13. IEEE Computer Society (2014)
3. Bruns, G., Huth, M.: Access-control policies via Belnap logic: effective and efficient composition and analysis. In: Proceedings of CSF 2008: 21st IEEE Computer Security Foundations Symposium, pp. 163–176 (2008)
4. Casadei, R., Aldini, A., Viroli, M.: Combining trust and aggregate computing. In: Cerone, A., Roveri, M. (eds.) SEFM 2017. LNCS, vol. 10729, pp. 507–522. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74781-1_34
5. Colombo, P., Ferrari, E.: Fine-grained access control within NoSQL document-oriented datastores. Data Sci. Eng. **1**(3), 127–138 (2016)
6. De Nicola, R., et al.: The SCEL language: design, implementation, verification. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems. LNCS, vol. 8998, pp. 3–71. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16310-9_1
7. Dwork, C., Roth, A.: The algorithmic foundations of differential privacy. Found. Trends Theor. Comput. Sci. **9**(3–4), 211–487 (2013)
8. Ercan Nergiz, M., Clifton, C.: δ-presence without complete world knowledge. IEEE Trans. Knowl. Data Eng. **22** (2010). https://ieeexplore.ieee.org/document/4912209/
9. Floriano, E., Alchieri, E., Aranha, D.F., Solis, P.: Providing privacy on the tuple space model. J. Internet Serv. Appl. **8**(1), 19:1–19:16 (2017)

10. Hall, R., Rinaldo, A., Wasserman, L.: Differential privacy for functions and functional data. J. Mach. Learn. Res. **14**(1), 703–727 (2013)
11. Hankin, C., Nielson, F., Nielson, H.R.: Advice from Belnap policies. In: Computer Security Foundations Symposium, pp. 234–247. IEEE (2009)
12. Kenig, B., Tassa, T.: A practical approximation algorithm for optimal k-anonymity. Data Min. Knowl. Disc. **25**(1), 134–168 (2012)
13. Li, N., Li, T., Venkatasubramanian, S.: t-closeness: privacy beyond k-anonymity and l-diversity. In: International Conference on Data Engineering (ICDE), pp. 106–115 (2007)
14. Lluch-Lafuente, A., Loreti, M., Montanari, U.: Asynchronous distributed execution of fixpoint-based computational fields. Log. Methods Comput. Sci. **13**(1) (2017)
15. Machanavajjhala, A., Kifer, D., Gehrke, J., Venkitasubramaniam, M.: l-diversity: privacy beyond k-anonymity (2014)
16. Margheri, A., Pugliese, R., Tiezzi, F.: Linguistic abstractions for programming and policing autonomic computing systems. In: 2013 IEEE 10th International Conference on and 10th International Conference on Autonomic and Trusted Computing (UIC/ATC) Ubiquitous Intelligence and Computing, pp. 404–409 (2013)
17. McSherry, F.: Privacy integrated queries: an extensible platform for privacy-preserving data analysis. Commun. ACM **53**(9), 89–97 (2010)
18. Mehta, A., Elnikety, E., Harvey, K., Garg, D., Druschel, P.: QAPLA: policy compliance for database-backed systems. In: 26th USENIX Security Symposium (USENIX Security 2017), Vancouver, BC, pp. 1463–1479. USENIX Association (2017)
19. Official Journal of the European Union. Regulation (EU) 2016/679 of the European parliament and of the council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/EC (general data protection regulation), L119, pp. 11–88, May 2016
20. Samarati, P., Sweeney, L.: Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. Technical report, Harvard Data Privacy Lab (1998)
21. Soria-Comas, J., Domingo-Ferrer, J., Sánchez, D., Martínez, S.: t-closeness through microaggregation: strict privacy with enhanced utility preservation. CoRR, abs/1512.02909 (2015)
22. Sweeney, L.: k-anonymity: a model for protecting privacy. Int. J. Uncertain. Fuzziness Knowl. Based Syst. **10**(5), 557–570 (2002)
23. Viroli, M., Damiani, F.: A calculus of self-stabilising computational fields. In: Kühn, E., Pugliese, R. (eds.) COORDINATION 2014. LNCS, vol. 8459, pp. 163–178. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43376-8_11