Defining, Enforcing and Checking Privacy Policies In Data-Intensive Applications

Michele Guerriero
Politecnico di Milano
Milano, Italy
michele.guerriero@polimi.it

Damian Andrew Tamburri
TU/e - JADS
Eindhoven, N.etherlands
d.a.tamburri@tue.nl

Elisabetta Di Nitto
Politecnico di Milano
Milano, Italy
elisabetta.dinitto@polimi.it

ABSTRACT

The rise of Big Data is leading to an increasing demand for large-scale data-intensive applications (DIAs), which have to analyse massive amounts of *personal* data (e.g. customers' location, cars' speed, people heartbeat, etc.), some of which can be *sensitive*, meaning that its confidentiality has to be protected. In this context, DIA providers are responsible for enforcing *privacy policies* that account for the privacy preferences of data subjects as well as for general privacy regulations. This is the case, for instance, of data brokers, i.e. companies that continuously collect and analyse data in order to provide useful analytics to their clients. Unfortunately, the enforcement of privacy policies in modern DIAs tends to become cumbersome because (i) the number of policies can easily explode, depending on the number of data subjects, (ii) policy enforcement has to autonomously adapt to the application context, thus, requiring some non-trivial runtime reasoning, and (iii) designing and developing modern DIAs is complex per se. For the above reasons, we need specific design and runtime methods enabling so called *privacy-by-design* in a Big Data context. In this article we propose an approach for specifying, enforcing and checking privacy policies on DIAs designed according to the Google Dataflow model and we show that the enforcement approach behaves correctly in the considered cases and introduces a performance overhead that is acceptable given the requirements of a typical DIA.

CCS CONCEPTS

• **Security and privacy** → **Privacy protections**; *Domain-specific security and privacy architectures*; • **Software and its engineering**;

KEYWORDS

Data Privacy, Dataflow Applications, Big Data, Context-Aware Privacy

ACM Reference Format:
Michele Guerriero, Damian Andrew Tamburri, and Elisabetta Di Nitto. 2018. Defining, Enforcing and Checking Privacy Policies In Data-Intensive Applications. In *SEAMS '18: SEAMS '18: 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3194133.3194140

1 INTRODUCTION

The huge and ever-growing amount of data continuously produced by the digital environment in which we are immersed, has led over the last few years to the emergence of many technologies enabling the exploitation of what is generally referred to as Big Data [8, 9]. For this reason, the way we design and develop data-intensive applications (DIAs), i.e., applications that devote most of their time to I/O and data manipulation operations, is experiencing significant changes moving from a classical batch processing model (see MapReduce [10]) to a dataflow model (the most prominent representative of this approach is the Google Dataflow model [1]) that allows to easily balance correctness of results, latency, and costs.

Nevertheless, there are still many open issues connected to Big Data, some of the most critical ones being related to data privacy. DIAs often process *sensitive* data, i.e. data whose confidentiality has to be preserved. According to [5], data privacy means not only to guarantee confidentiality, that is, data protection from *unauthorized access*, but also to take into account policies coming from *legal privacy regulations* as well as *individual privacy preferences*. Such *privacy policies* are in many cases *context-based* as their enforcement should adapt based on certain contextual conditions [2, 13, 16, 18], for example, the location of the person accessing data, for which purpose he/she is performing the access, etc. This further adds complexity to the already complex development of large-scale DIAs.

The research literature has clearly manifested a need for more research on how to effectively and efficiently provide privacy-awareness in modern DIAs [5]. On the one hand, data subjects need to be empowered with more control over how their data is collected and used, while currently, in most cases, they can just agree or not to a given (and often not intuitive) privacy contract. On the other hand, simplified methods are needed to reduce the heavy investments now required for privacy-specific coding and testing of DIAs.

This paper focuses on the second part of the problem by proposing a novel framework to model and enforce privacy policies in modern DIAs. We assume that DIAs are modeled as *dataflow* applications managing a set of *data streams*, some of which may carry *sensitive* information and be externally *observed* (by humans or other applications). The way in which such data streams are observed should adapt over time, depending on the evolution of certain *contextual conditions* and on the *privacy policies* that have been defined for the specific pieces of data being transferred. These policies may encode privacy regulations as well as the privacy preferences of the many data subjects whose data are processed by the DIA. Their enforcement should result in hiding or not some pieces of data, as specified by the data owner. Along this direction, the contributions of this paper are:

- A model and a language for specifying privacy policies on dataflow applications; the semantics of such policies is formalised using a Metric First Order Temporal Logic (MFOTL), which enables the runtime verification of correct policy enforcement.
- A policy enforcement algorithm, which works by means of dataflow rewriting, i.e. creating a new privacy-aware dataflow out of an original one algorithmically. Our framework instruments a dataflow application by introducing some operators that 1) **M**onitor the data that are needed to **A**nalyse the occurrence of contextual conditions and 2) based on the rules (**P**lanning) defined by privacy policies, **E**xecute privacy-specific self-adaptation actions.
- A prototype implementation of our framework on a distributed dataflow engine along with its evaluation.

The remainder of this paper is organised as follows. Section 2 provides background information on the dataflow model; Section 3 presents a motivating scenario and states the problem; Section 4 introduces our model for privacy policies in dataflow applications; Section 5 describes our policy enforcement strategy; Section 6 reports on our experiments to assess correctness and performance of the approach; Section 7 discusses related work; Section 8 discusses future work and concludes the paper.

2 BACKGROUND

In *dataflow computing* an application is seen as a directed graph in which edges represent data streams and nodes represent (1) *functional operators* (map, reduce, filter, etc.), which transform some input streams to produce one or more output streams, (2) *data sources*, which produce data streams, or (3) *data sinks*, which output a data stream to some external system. Operators can be stateful, meaning that they keep an internal state (e.g. aggregation operators) which is supposed to be transparently managed and maintained by the application runtime (i.e. the execution engine). Each data stream within a dataflow application can be essentially seen as an infinite and append-only table of events (or *tuples*) which is continuously updated. Tuples in a data stream are typically ordered by means of a *timestamp*, which is the time at which a tuple arrives to, or is produced by, the application. Moreover, a tuple of a data stream is typically structured into a finite set of fields, which essentially define the *schema* of the stream. The computational model is inherently parallel and well suited for distributed and large-scale data processing. In fact, being the functional operators independent from each other, they can execute concurrently (potentially on different machines) as soon as their respective input streams are available, thus enabling *task parallelism. Data parallelism* can also be exploited by means of data partitioning, i.e. splitting a given data stream into disjoint partitions, each of which is then assigned to a given replica of an operator in order to be processed. Finally, in order to deal with the infiniteness of data streams, dataflow engines typically provide a notion of *windowing*, i.e. chopping up a data stream into finite pieces along some temporal boundaries. The dataflow engine creates and populates windows according to the timestamps of the incoming tuples and to different windowing strategies (fixed, sliding, etc.). Windows essentially allow to properly capture the evolution of data over time.

The *Google Dataflow model* is the most relevant representative of the dataflow model in the context of large-scale DIAs. It has been specifically designed around the idea that "we live in a world of unbounded, unordered and massive-scale datasets and that we need to look and them as such, without making any assumption on the eventual completeness of a dataset" [1]. Essentially, the Google Dataflow Model identifies data streams as a first-class citizen in the design of modern DIAs and proposes an approach to deal with their characteristics (e.g. out-of-orderness).

3 MOTIVATING SCENARIO AND PROBLEM STATEMENT

As a motivating example, we consider the case of an e-commerce company, let's call it *GreatSeller*, which is developing a DIA that continuously processes consumers' financial transactions and updates statistics, some of which can be observed in real-time by external users and applications. In fact, *GreatSeller* wants to sell statistics about consumers, thus acting also as a data broker. In particular, *GreatSeller*'s DIA continuously publishes the following statistics:

- **statistics1**: the total amount of money spent by each consumer during the last 10 minutes.
- **statistics2**: the number of transactions issued by each consumer during the last 10 minutes.
- **statistics3**: the number of consumers who have spent more than \$1000 during the last hour.

Let us assume that the marketing company *MarketConsult* has bought from *GreatSeller* real-time access to **statistics1** **statistics2** with the *purpose* of doing analytics and that *GreatSeller* has informed all the consumers of which it collects data. Since the DIA consumes and publishes sensitive data or data that is computed starting from sensitive inputs, *GreatSeller* is supposed to properly protect the privacy of individuals by considering their privacy preferences as well as general privacy regulations. According to Nissenbaum [16], privacy is provided by appropriate information flows, which conform to certain *contextual conditions*. The latter can be defined in terms of facts that *hold at the time the access occurs*, such

as the identity of the user or the purpose of the access, as well as facts that *have occurred at some point in the past*. Ideally, a data broker should be able to control the contextual conditions that dictate how data flows to the users.
In particular, let us also assume that a specific consumer, Bob, has specified the following two privacy policies:

- $Policy_1$: Bob agrees on letting *MarketConsult* know how much he has spent over the last 10 minutes, as this could result in the formulation of some interesting offers. At the same time, he does not want *MarketConsult*'s employees to know if he has spent more than \$100, if it happened at some point during the last 30 mins that the value of **statistics2** for Bob went over 3. In this specific case, Bob asks *GreatSeller* to lower any higher amount to \$100.
- $Policy_2$: When *MarketConsult* observes the application, Bob does not allow *GreatSeller* to include his data in the computation of the number of consumers spending more than \$1000 in one hour.

By violating the policies set by Bob, or those set by all other consumers of which *GreatSeller* collects data, the broker could incur in serious penalties, from fines (ranging from a few hundred to millions of dollars) till the closure of the company.

The described DIA could be developed as the dataflow application depicted in Figure 1, which manages the following data streams:

- S_1: the input stream of transactions, which is produced by data source $TRXSRC$ and conforms to the schema *(transactionId: int, dataSubject: string, amount: double, recipientId: string)*.
- S_2: the stream containing for each consumer the number of issued transactions over the last 10 minutes; it is produced by operator OP_1 by applying a time window of 10 minutes on S_1 and conforms to the schema *(dataSubject:string, nTransactions: int)*
- S_3: the stream containing for each consumer the total amount spent over the last 10 minutes; it is produced by operator OP_2 by applying a time window of 10 minutes on S_1 and conforms to the schema *(dataSubject: string, totalAmount: double)*
- S_4: the stream containing the number of consumers that have spent more than \$1000 over the last hour; it is produced by operator OP_3 by applying a time window of 1 hour on S_3 and conforms to the schema *(nTopUsers: int)*.

Of these, S_2, S_3 and S_4 are observable. They are different from a privacy viewpoint: while data in streams S_2 and S_3 refer to specific consumers (i.e. given a consumer, an external user can observe the evolution of her statistics), single data points in stream S_4 represent aggregations of data concerning all the consumers. Therefore, a consumer can have a different type of control on her data within each type of stream: while in the first case a consumer may want to directly control if (i.e. in which context) or how her own data are observed, in the second case, she can only decide if her data should take part or not in the computation that produces the statistics.

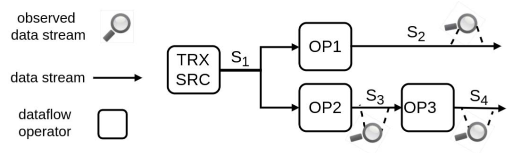

Figure 1: Example dataflow application for processing consumers' transactions.

To efficiently implement the enforcement of multiple policies similar to $Policy_1$ and $Policy_2$ within the same dataflow application can become a difficult task, since (i) the number of policies can easily explode, depending on the number of data subjects, (ii) the policy enforcement has to autonomously adapt to the application context and, thus, it requires some non-trivial runtime reasoning (consider for instance $Policy_1$ which requires knowledge about the number of issued transactions during the last 30 mins) and (iii) designing and developing modern DIAs is a complex activity per se, which requires specific skills that are not easy to find on the market [11][12].

Overall, we need a way to simplify the definition and enforcement of context-based privacy policies in dataflow applications. In this paper we tackle this problem. We assume data subjects define explicitly their personalized privacy policies. Nevertheless, our approach could be also used to specify policies coming from privacy regulations. We do not offer an end-user policy language for the moment, but we assume the policies are translated into a simple, machine-oriented language that we present in Section 4. Our tool automatically rewrites the original code of the dataflow application to incorporate the mechanisms needed to enforce the defined policies (Section 5). The rewritten code exploits the dataflow nature of the application and inserts connections to new operators that enact the enforcement of policies. By having a simple language for specifying privacy policies and an automatic mechanism for their enforcement our we attempt to mitigate the three problems mentioned above. The current prototype works on applications implemented in Apache Flink [6], but its conceptual ground would remain valid also for implementations exploiting a different dataflow framework.

4 PRIVACY POLICIES IN DATAFLOW APPLICATIONS

In this Section we introduce our model for privacy policies. Given a dataflow application A, we denote with S_A the set of data streams A produces and with DS the set of data subjects whose sensitive data are consumed by A. Each data stream $s \in S_A$ is structured into a finite and totally ordered set F_s of fields, denoted by $\langle s.f_1, ..., s.f_{|F_s|} \rangle$. Given a tuple $t \in s$, we denote with $t.f_i$ the value of $s.f_i$ in t. We distinguish between three disjoint sets of data streams:

- the set of *subject-specific* streams $S_{sp} \subseteq S_A$. A data stream $s \in S_{sp}$ is such that each tuple $t \in s$ refers to a specific data subject by means of an additional field $t.dataSubject \notin F_s$.

- the set of *subject-generic* streams $S_g \subseteq S_A$. A data stream $s \in S_g$ is such that (i) each tuple $t \in s$ does not refer to a specific data subject and (ii) s is produced by an operator with at least an input stream $s_{in} \in S_{sp}$.
- the set of *non-personal* streams $S_{np} \subseteq S_A$. A data stream $s \in S_{np}$ is such that (i) each tuple $t \in s$ does not refer to a specific data-subject and (ii) s is produced by an operator with no input stream in S_{sp}. In practice, such data streams are not exposed to any privacy risk, so we will not consider them in our privacy model.

Referring to the scenario of Section 3, S_1, S_2, and S_3 belong to S_{sp} while S_4 to S_g. As informally discussed, they need different types of privacy preserving policies, since the kind of control a data subject can apply over them differs. In the next two subsections we introduce the two types of policies that our model provides for the two sets S_{sp} and S_g of data streams.

4.1 View Creation Policies (VCPs)

Given a subject-specific stream $s \in S_{sp}$, let $t \in s$ be a tuple that refers to a data subject $ds \in DS$. ds should precisely specify *when* (the context) and *how* (how much information of) the tuple can be observed. As for the latter, the amount of information to be disclosed can be tuned by defining a *view* over the set of fields F_s carried by s. Such view decreases to a certain extent the amount of information contained in t, for instance, transforming specific numeric values into ranges or into a fixed value (for instance, in our example any amount greater than \$100 can become equal to this value).

View Creation Policies (VCPs) allow data subjects to bind views of a subject-specific stream to different contextual conditions. To construct views we rely on the concept of *Domain Generalization Hierarchy* (DGH), as defined and used in the field of data anonymization. According to Sweeney [20], given a table T and a field f of T (i.e. a column of T), whose domain of values is denoted by D_f, a DGH for f is an ordered set of domains $D_0 \rightarrow ... \rightarrow D_n$ and functions $h_1 \rightarrow ... \rightarrow h_n$ such that $h_i : D_{i-1} \mapsto D_i$, where $D_0 = D_f$. Figure 2 shows a possible DGH defined over a field with domain $\mathbb{N}$, where $h_1 : \mathbb{N} \mapsto D_1$ lowers each natural number greater than 100 to 100, while $h_2 : D_1 \mapsto D_2$ maps each element in D_1 to 50. Given a value $v \in D_f$ we say that $h_k\left(h_{k-1}\left(...h_1\left(v\right)...\right)\right)$ is the *k-generalization* of v, where parameter k indicates the number of generalization steps that have to be applied in order to obtain the desired view of v. For instance, given the value $v = 104$ for field f, $h_1\left(v\right) = 100$ is its 1-generalization, while $h_2\left(h_1\left(v\right)\right) = 50$ is its 2-generalization. We denote the k-generalization of a value v for field f with $gen^{k_f}\left(v\right)$.

The concept of DGH is immediately applicable to our data streams as we see them as infinite and append-only tables. Thus, our model defines the concept of *Generalization Vector* (GV) as follows: given a data stream $s \in S_{sp}$ with fields $\langle s.f_1...s.f_{|F_s|}\rangle$ and corresponding $\langle DGH_1...DGH_{|F_s|}\rangle$, a Generalization Vector $\overline{gv} \in \mathbb{N}^{|F_s|}$ defines the parameter

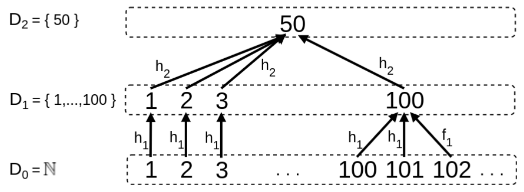

Figure 2: Example Domain Generalization Hierarchy over $\mathbb{N}$.

k_i to be used for generalizing each value of $s.f_i$. We denote with GV_s the set of all possible generalization vectors for s. In our model we assume the DIA provider to be responsible for defining DGHs for all the sensitive fields contained in the various subject-specific streams. A View Creation Policy, instead, allows data subjects to specify for a given field the desired k-generalization to be used, according to the DGH defined for that field. From the above premises, a VCP is then defined as in Formula 1, where c is the contextual condition that enables the policy and is expressed by means of our condition language L (see Section 4.3). Informally, the semantics of a VCP is the following: given a data stream $s \in S_{sp}$ and a tuple $t \in s$ referring to the data subject $ds \in DS$, if the contextual condition c holds at the time t is received, then the view $\langle gen^{k_{f_1}}\left(t.f_1\right), ..., gen^{k_{f_{|F_s|}}}\left(t.f_{|F_s|}\right)\rangle$ must be created and emitted instead of t. This semantics is formalized in Section 4.5.

$$\begin{aligned} VCP = \{(s, ds, c, g) :\ & s \in S_{sp},\ ds \in DS, \\ & c\ is\ an\ expression\ of\ L,\ g \in GV_s\}. \end{aligned} \tag{1}$$

4.2 Data Subject Eviction Policies (DSEPs)

Given a subject-generic stream $s \in S_g$, a tuple $t \in s$ does not directly refer to a specific data subject. Still, the dataflow operator OP that produces s takes as input a data stream (or multiple data streams) $s_{in} \in S_{sp}$. In this situation, a data subject $ds \in DS$ may want to specify when she does not want her tuples in s_{in} (or in all the subject-specific stream that are input of OP) to be considered by operator OP. This is the goal of what we call *Data Subject Eviction Policies* (DSEP). More formally, a DSEP is defined as in Formula 2, where c is the contextual condition that enables the policy and is expressed by means of our condition language L (see Section 4.3). Informally, the semantics of a DSEP is the following: given a data stream $s \in S_g$, all the tuples referring to a data subject $ds \in DS$ must be evicted from the input of the operator that produces s, whenever the contextual condition c holds. This semantics is formalized in Section 4.5.

$$\begin{aligned} DSEP = \{(s, ds, c) :\ & s \in S_g,\ ds \in DS, \\ & c\ is\ an\ expression\ of\ L\}. \end{aligned} \tag{2}$$

4.3 The L Language for Contextual Conditions

Both VCPs and DSEPs are enabled by some *contextual conditions*. Our model includes a language called L for expressing such conditions on a set of *contextual variables*.

For a dataflow application A, the set of *contextual variables* is defined as $X_A = \{(\bigcup_{s \in S_A} F_s) \bigcup SCV\}$, where $SCV = \{observerId, role, purpose\}$. X_A is essentially the union of all the fields provided by all the data streams produced by the application, plus a set of *static contextual variables* (SCV), i.e., the properties of application observers that are relevant from a privacy point of view, according to the privacy related literature [2] [15].

The proposed language L, is a derivative of a Metric Temporal Logic (MTL) [14]. In particular, we make use of the $Once_{[T',T'']}$ operator, which is the past version of $Eventually_{[T',T'']}$. T' and T'' represent a delay in the past from the current instant in terms of the implicit time unit. Time intervals T' and T'' are finite and either open or closed, both on the left edge T' and right edge T''. The generic formula $Once_{[T',T'']}\ \phi$ is true at time instant t when there exists a time instant $t' \in [t - T', t - T'']$ where ϕ holds. Condition in L are then defined as follows:

Definition 4.1. Let A be a dataflow application and X_A the corresponding set of contextual variables; each variable $x \in X_A$ has a domain of possible values, denoted as D_x; if x is a categorical variable, D_x is provided with the set $OP_c = \{=, \neq\}$ of relational operators; if x is a numerical variable, then D_x is provided with the set $OP_n = \{=, \neq, >, <, \geq, \leq\}$ of relational operators. An *atomic condition* c defined over $x \in X_A$ has the form (x *op* v) where $v \in D_x$ and *op* $\in OP_c$ if x is a categorical variable, *op* $\in OP_n$ otherwise. The conditions of L over X_A are defined as follows:

- An atomic condition is a condition of L
- Let c be an atomic condition of L defined over a variable $x \in X_A$; then $Once_{[T',\ T'']}\ c$ is a condition of L. We say this kind of condition to be a *past condition*.
- Let c_i and c_j be conditions of L; then $c_i \wedge c_j$ is a condition of L.

Essentially, the fields of the various data streams, plus the set of *static contextual variables*, are considered as the application variables and the language allows policies to predicate over the last value of these variables as well as over the occurrence of past values within a given time window.

The evaluation of a condition varies depending on the type of stream over which it is applied. Given a data stream $s \in S_A$ and an atomic condition $c = (x\ op\ val)$, such that $x \in F_s$: if $s \in S_{sp}$, then c is evaluated considering only those tuples in s that refer to the data subject who specifies the condition, else if $s \in S_g$, then c is evaluated considering all the tuple of s. This is a consequence of the fact that a data stream $s \in S_g$ is unique for all the data subjects, while for a data stream $s \in S_{sp}$ each data subject has and controls her own partition of s (i.e. all those tuples $t \in s$ that refer to her).

4.4 Policy Examples

Now that we have a language for specifying privacy policies on dataflow applications, we can formalize the two example policies introduced in Section 3.

In particular, let us assume that the DGH shown in Figure 2 is attached to field $S3.totalAmount$ and that *static contextual variables* are fed to the application by means of an additional data stream, SCV, conforming to the schema *(obsId: string, role: string, purpose: string)*, so that contextual conditions on such variables can be evaluated. Although our current prototype relies on this assumption (as will be clear in Section 5), a different implementation could have used a different way of making the application aware about static contextual variables.

Then, $Policy_1$ can be written as the VCP shown in Formula 3, which specifies that, given a tuple $t \in S_3$ which refers to Bob, if (i) the S_3 observer is an employee of *MarketConsult* with the purpose of doing analytics and (ii) there has been a tuple in $t' \in S_2$ during the last 30 minutes such that $t'.nTransactions$ was greater than 3 and (iii) the field $t.amount$ is greater than 100, then the generalization vector $(0, 1)$ has to be applied to t, i.e., the field $t.dataSubject$ has to be left as it is, while the field $t.totalAmount$ has to be generalized to level 1 of its DGH.

$Policy_2$ can be written as the DSEP shown in Formula 4, which specifies that if someone from *MarketConsult* is observing S_4 with the purpose of doing some analytics, then tuples referring to Bob have to be evicted from the input of the operator that produces S_4.

$$\begin{aligned} VCP_{Policy_1} = (&S_3,\ Bob,\ obsId = MarketConsult, \\ &\wedge\ purpose = analytics\ \wedge\ role = employee \\ &\wedge\ Once[30,0]\ S_2.nTransactions > 3 \\ &\wedge\ S_3.amount > 100,\ (0,1)) \end{aligned} \tag{3}$$

$$\begin{aligned} DSEP_{Policy_2} = (&S_4,\ Bob,\ obsId = MarketConsult, \\ &\wedge\ purpose = analytics) \end{aligned} \tag{4}$$

4.5 Defining the Semantics of Privacy Policy Enforcement

The enforcement of privacy policies defined on a stream s has to ensure that, when the conditions expressed in the policies occur, the stream fields are observed according to the policy requirements. For VCPs this consists in the application of the proper generalization on fields in s, for DSEP this corresponds to the eviction of some tuples from the input of the operator that produces s.

Consider, for instance, the scenario is Figure 3 that refers to a possible snapshot of the running example. In particular, the figure shows the evolution of streams S_3 and S_2 and of the static contextual variables (i.e., the SCV data stream). These streams are those relevant to the enforcement of $Policy_1$. The last tuple of SCV indicates that a *MarketConsult* employee is observing the streams for the purpose of analytics. On stream S_2 we see flowing information about the number of transactions performed by Bob, May and John. Bob, in particular, has associated two tuples, one of which is of interest for $Policy_1$ as it is stating that Bob has performed 7 transactions (see the bold tuple in the figure). This tuple

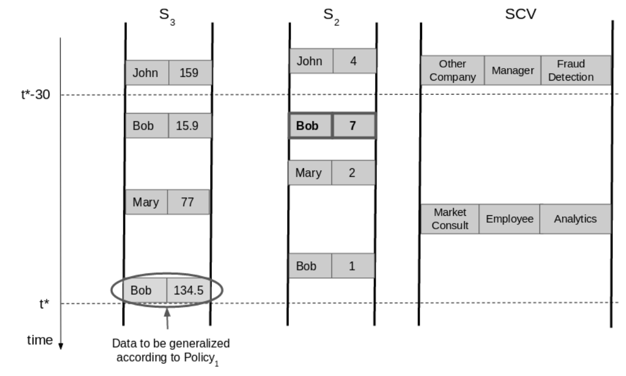

Figure 3: Possible evolution of streams in the running example.

has occurred in the time window from $t^* - 30$ to t^*. This implies that a correct enforcement of $Policy_1$ should generalize the last tuple on stream S_3 before making it observable by the *MarketConsult* employee, given that the corresponding amount is higher than \$100.

To formalize these enforcement rules we adopt a Metric First Order Temporal Logic (MFOTL) [4].

For subject-specific streams protected by VCPs, the enforcement rule distinguishes between the *original* value and the mutated one. In particular, for any field a of a subject-specific stream, we indicate with a_{org} its original value and with a_{obs} its observed value. This last one can be mutated, depending on the policy. For subject-generic streams protected by DSEPs we do not need to keep such distinction as a field is either considered or not considered by a certain operator.

Formulae 5 and 6 represent generic instances of VCPs and DSEPs respectively.

$$\begin{aligned} VCP_{gen} = (ds,\ s,\ & Once[T_1', T_1''] \ f_1 \ OP_{f_1} \ v_{f_1} \\ & \wedge \ldots \wedge \ Once[T_p', T_p''] \ f_p \ OP_{f_p} \ v_{f_p} \\ & \wedge \ f_q \ OP_{f_q} \ v_{f_q} \ \wedge \ \ldots \wedge \ f_w \ OP_{f_w} \ v_{f_w}, \\ & (k_{a_1},\ \ldots\ ,\ k_{a_n})) \end{aligned} \tag{5}$$

$$\begin{aligned} DSEP_{gen} = (ds,\ s,\ & Once[T_1', T_1''] \ f_1 \ OP_{f_1} \ v_{f_1} \\ & \wedge \ldots \wedge \ Once[T_p', T_p''] \ f_p \ OP_{f_p} \ v_{f_p} \\ & \wedge \ f_q \ OP_{f_q} \ v_{f_q} \ \wedge \ \ldots \wedge \ f_w \ OP_{f_w} \ v_{f_w}) \end{aligned} \tag{6}$$

Let us begin with VCP_{gen}, which is applied by data subject ds on the subject-specific stream $s = \langle dataSubject,\ a_1, \ldots,\ a_n \rangle$ produced by application A and where (i) $f_1, \ldots, f_p$ are fields of other data streams within the application over which the policy specifies **past** conditions, (ii) $f_q, \ldots, f_w$ are fields of other data streams within the application over which the policy specifies **atomic** conditions and (iii) $(k_{a_1}, \ldots, k_{a_n})$ is the generalization vector to be applied over s, with $k_{a_i} \in \mathbb{N}$. The corresponding MFOTL Formula 7 defines the VCP-type semantics in this case. In particular, it states that, given a tuple $t_s \in s$, which refers to the data subject ds owner of VCP_{gen}, if the expected generalization of the original value of one of t fields does not correspond to its observed value, then the condition specified in VCP_{gen} must be false, that is, VCP_{gen} is not enforced.

Let us now consider $DSEP_{gen}$ shown in Formula 6, which is specified by a data subject ds on a subject-generic stream $s = \langle a_1, \ldots,\ a_n \rangle$. Let us call $s_1,\ \ldots,\ s_n$ the subject-specific streams that are input to the operator producing s. For each $s_i = \langle dataSubject,\ a_1, \ldots,\ a_{n_i} \rangle$ the MFOTL Formula 8 must hold in order to the DSEP to be satisfied. It states that, if there is a tuple $t_{s_i} \in s_i$, which refer to the data subject ds owner of $DSEP_{gen}$, then the condition specified in VCP_{gen} must be false, that is, $DSEP_{gen}$ is not enforced.

Specific instances of such MFOTL formulae can be automatically checked using available tools, allowing us to monitor the correctness of the policy enforcement, as we will discuss in Section 6.1.

$$\begin{aligned} & t_s(dataSubject,\ a_{1_{obs}},\ \ldots\ ,\ a_{n_{obs}},\ a_{1_{org}},\ \ldots,\ a_{n_{org}}) \\ & \wedge \ dataSubject = \ ds \ \wedge \ (\neg \ (gen^{k_{a_1}}(a_{1_{org}}) \ = \ a_{1_{obs}}) \ \vee \ \ldots \\ & \vee \ \neg \ (gen^{k_{a_n}}(a_{n_{org}}) \ = \ a_{n_{obs}})) \ implies \ \neg \ (\\ & Once[T_1',\ T_1''] \ f_1 \ OP_{f_1} \ v_{f_1} \ \wedge \ \ldots \\ & \wedge \ Once[T_p',\ T_p''] \ f_p \ OP_{f_p} \ v_{f_p} \\ & \wedge \ Once[*,\ 0] \ f_q \ OP_{f_q} \ v_{f_q} \\ & \wedge \ \neg\exists \ v_{f_q}'.(\neg \ f_q \ OP_{f_q} \ v_{f_q}') \ Since[*,\ 0] \ f_q \ OP_{f_q} \ v_{f_q} \ \wedge \ \ldots \\ & \wedge \ Once[*,\ 0] \ f_w \ OP_{f_w} \ v_{f_w} \\ & \wedge \ \neg\exists \ v_{f_w}'.(\neg \ f_w \ OP_{f_w} \ v_{f_w}') \ Since[*,\ 0] \ f_w \ OP_{f_w} \ v_{f_w} \ ) \end{aligned} \tag{7}$$

$$\begin{aligned} & t_{s_i}(dataSubject,\ a_1,\ \ldots,\ a_{n_i}) \\ & \wedge \ dataSubject = \ ds \ \implies \ \neg \ (\\ & Once[T_1',\ T_1''] \ f_1 \ OP_{f_1} \ v_{f_1} \ \wedge \ \ldots \\ & \wedge \ Once[T_p',\ T_p''] \ f_p \ OP_{f_p} \ v_{f_p} \\ & \wedge \ Once[*,\ 0] \ f_q \ OP_{f_q} \ v_{f_q} \\ & \wedge \ \neg\exists \ v_{f_q}'.(\neg \ f_q \ OP_{f_q} \ v_{f_q}') \ Since[*,\ 0] \ f_q \ OP_{f_q} \ v_{f_q} \ \wedge \ \ldots \\ & \wedge \ Once[*,\ 0] \ f_w \ OP_{f_w} \ v_{f_w} \\ & \wedge \ \neg\exists \ v_{f_w}'.(\neg \ f_w \ OP_{f_w} \ v_{f_w}') \ Since[*,\ 0] \ f_w \ OP_{f_w} \ v_{f_w} \ ) \end{aligned} \tag{8}$$

5 AUTOMATIC POLICY ENFORCEMENT

Our policy enforcement prototype is developed in Apache Flink as an open-source library [1]. DIA developers exploit the standard Flink Streaming APIs for defining the dataflow application and the API offered by our library to define policies. When policies are defined, the library applies the rewriting algoritm that transforms the initial DIA into one that is augmented with *privacy-enhancing dataflow operators* that enforce the execution of privacy policies. We say that the result of rewriting is a *privacy-aware* version of the DIA.

At runtime, it is the DIA itself that realizes the policy enforcement, automatically self-adapting to evolving contextual conditions. In particular, depending on the observer and on the specific events that happen during the DIA execution, the application adapts its output by exploiting generalization or eviction of fields.

[1] https://github.com/MicheleGuerriero/dia-privacy-library

The underlying Flink engine allows to easily parallelize the executions of our privacy-enhancing operators. To improve performance, at runtime, different instances of an operator may exist, each of which processes a different partition of a data stream. Partitions are defined based on data subjects, meaning that, for example, tuples of a data stream that refer to two distinct data subjects may be processed by two different replicas of an operator. The DIA developer has only to specify the degree of parallelism to be used for the privacy enforcement.

5.1 Privacy-Enhancing Dataflow Operators

The privacy-enhancing dataflow operators are the basic building blocks that we combine to enforce privacy policies on specific streams. There are four types of operators that are described in the following of this section.

StaticContextSource (SCSRC): this is data source operator that provides the dataflow application with updates about the static contextual variables. It produces a data stream SCV such that at each point in time the last emitted tuple contains the most recent value for each variable $x \in SCV$.

PastConditionChecker (PCC): it is an operator responsible for checking the validity of past conditions on a given data stream s. More specifically, given a set of past conditions to be verified on s: $Once[T'_1, T''_1]\ c_1,\ \ldots\ ,\ Once[T'n,$ $T''_n]\ c_n$, a PCC keeps a sliding window that at time $\hat{T}$ contains all the tuples received between $\hat{T} - max(T'_1, ..., T'_n)$ and $\hat{T} - min(T''_1, ..., T''_n)$. When a new tuple $t \in s$ arrives, the PastConditionChecker updates the sliding window, according to the timestamp of t, and re-evaluates all the past conditions against the updated content of the window. If s is a subject-specific stream, then the evaluation of the past conditions is performed considering only those tuples in the window that refer to the data subject who specified the condition, else if s is a subject-generic stream, then all the tuples in the window are considered (according to how past conditions have been defined in Section 4.3). Finally, the PastConditionChecker emits a new tuple in its output stream which reports the current truth value for each managed past condition. A PCC essentially decouples the evaluation of past conditions from the actual enforcement of a policy.

ViewBuilder (VB): it is an operator responsible for the enforcement of VCPs applied on a given subject-specific stream s. The ViewBuilder takes as input the data stream s to be protected, along with the data streams needed to evaluate the atomic conditions occurring in the managed VCPs. For the evaluation of all past conditions required by the managed VCPs, the ViewBuilder is also connected to the output stream of the corresponding PCCs from which it receives the current truth values. The ViewBuilder produces a single output stream $s*$, which is the generalized view of s, according to the managed VCPs. $s*$ becomes the observable stream.

DataSubjectEvictor (DSE): it is an operator that takes part into the enforcement of the DSEPs applied on a given subject-generic stream s. This operator has to filter out private data before they arrive to the application-specific operator that generates s. Let O be this application-specific operator and $S_{o,in} = \{s_{in} | s_{in}\ input\ to\ O\}$ be the set of all subject-specific streams entering into O. Each s_{in} is taken as an input by a different DataSubjectEvictor operator together with (1) the output stream of all the PCCs that evaluate past conditions required by the managed DSEPs and (2) the data streams required to directly (i.e. by the DSE) evaluate the atomic conditions required by the managed DSEPs. The evictor produces a single output stream $s_{in-evict}$, which is the evicted version of s_{in} and that becomes the new input stream of the operator that produces s. In this case the original stream s remains the observed one, as the privacy enforcement is performed before it is produced.

5.2 Using Operators in the Running Example

Privacy-enhancing operators are to be incorporated into dataflow applications to enforce privacy policies. The operators together constitute a distributed MAPE loop, where some operators monitor certain data streams and analyse the occurrences of specific events, while others take decisions on how to adapt the application (based on the defined privacy policies) and execute privacy-specific adaptation actions.

Figure 4 shows how they are inserted in the example of Section 3 to enforce VCP_{Policy_1} and $DSEP_{Policy_2}$. The added operators and data streams are marked with a dotted line. Since VCP_{Policy_1} on S_3 contains a past condition over S_2, a PastConditionChecker takes S_2 as an input. Its output is sent to the ViewBuilder, which takes as input also the stream to be protected, S_3, and the stream SCV, since VCP_{Policy_1} contains an atomic condition over some static contextual variables. For each incoming tuple in S_3, the ViewBuilder evaluates if the condition specified by VCP_{Policy_1} holds, eventually generalizing the tuple according to the specified generalization vector. The output stream of the ViewBuilder, S'_3, becomes the new observed stream. In the case of multiple policies from different data subjects specified on S_3, the execution of both the PastConditionChecker and the ViewBuilder can be parallelized, with tuples from different data subjects being processed by different replicas.

Concerning $DSEP_{Policy_2}$, the operator that produces S_4 takes as input a single subject-specific stream, i.e. S_2; consequently, a single DataSubjectEvictor needs to be in place to take as input the stream S_2 to be evicted, as well as the SCV, since $DSEP_{Policy_2}$ contains an atomic condition over some static contextual variables. For each incoming tuple of S_2, the DataSubjectEvictor evaluates if the condition specified by $DSEP_{Policy_2}$ holds, eventually avoiding the emission of the tuple. The output stream of the DataSubjectEvictor becomes the new input stream for the operator that produces the observed stream S_4. Also the DataSubjectEvictor can

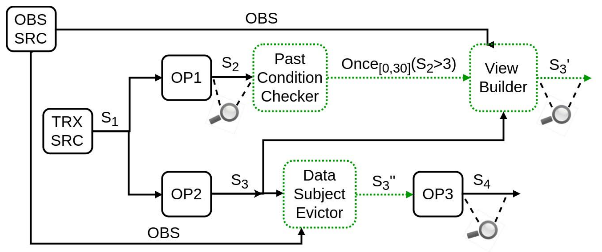

Figure 4: The running example after the execution of the rewriting algorithm.

be parallelized in the case of multiple policies. Such *privacy-aware* version of our running example can be automatically obtained using the algorithm presented in the next Section.

5.3 Dataflow Privacy-Rewriting Algorithm

In this Section we present our algorithm to automatically produce a privacy-aware version of a given DIA over which a number of privacy policies has been set. The algorithm pseudo-code is reported in Algorithm 1.

At the beginning, the `dataflowPrivacyRewriting` adds a $SCSRC$ data source, to provide the application with the set of static contextual variables. Then, the procedure starts looking for subject-specific streams *sp* for which there exists at least one VCP and, for each such stream, a new ViewBuilder operator is added to the application, which takes as input *sp* along with the set of all the VCPs defined over *s* (lines 3-6). Then, *sp* is removed from the inputs of its *consumers* (i.e. all the operators in the application that take *sp* in input) and is replaced with the output of the added ViewBuilder (lines 7-10). Then, the procedure looks for each subject-generic stream *sg* that has associated at least one DSEP. For each such stream and for each subject-specific stream s_{in} which is an input of the operator that produces *sg*, a DataSubjectEvictor is added to the application and wired to s_{in}. The latter will be removed from the inputs of the application-specific operator producing *sg* and replaced with the output of the added DataSubjectEvictor (lines 14-24).

Finally, for each ViewBuiler/DataSubjectEvictor operator that has been added, both the `addPastConditionCheckers` and the `addInputsForAtomicConditions` sub-procedures are invoked. The first one is responsible for adding all the Past-CondtionChecker operators needed by a privacy-enhancing operator *o*. More specifically, for each data stream s' in the current application, the sub-procedure checks if *o* requires a past condition to be verified over s' in order to evaluate one of the policies it is responsible for. If yes, a PastConditionChecker operator is added to the application, which takes as input s' and whose output is added to the inputs of *o*. Finally the past conditions defined by the various policies over s' managed by *o* are passed to the added Past-ConditionChecker in order to be checked at runtime. The `addInputsForAtomicConditions` sub-procedure is responsible for adding to the inputs of a privacy-enhancing operator *o*

Algorithm 1 Pseudo-code of the rewriting algorithm used to produce privacy-aware DIAs.

```
1:  procedure dataflowPrivacyRewriting(A)
2:      $privacyEnhancingOperators = \emptyset$
3:      $A.addStaticContextSource()$
4:      for all $sp \in A.S_{sp}$ do
5:          if $\neg sp.VCPs.isEmpty()$ then
6:              $vb = A.addVB(sp.VCPs)$
7:              $vb.addInput(sp)$
8:              for all $o \in s.consumers$ do
9:                  $o.removeInput(sp)$
10:                 $o.addInput(vb.outputStream)$
11:             end for
12:             $privacyEnhancingOperators.add(vb)$
13:         end if
14:     end for
15:     for all $sg \in A.S_g$ do
16:         if $\neg sg.DSEPs.isEmpty()$ then
17:             for all $s_{in} \in sg.producer.inputStreams \cap A.S_{sp}$ do
18:                 $dse = A.addDSE(sg.DSEPs)$
19:                 $dse.addInput(s_{in})$
20:                 $sg.producer.removeInput(s_{in})$
21:                 $sg.producer.addInput(dse.outputStream)$
22:                 $privacyEnhancingOperators.add(vb)$
23:             end for
24:         end if
25:     end for
26:     for all $o \in privacyEnhancingOperators$ do
27:         $addPastConditionCheckers(A, 0)$
28:         $setAtomicConditions(A, o)$
29:     end for
30: end procedure
31:
32: procedure addPastConditionCheckers(A, o)
33:     for all $s' inA$ do
34:         if $o.requiresPastConditionOn(s')$ then
35:             $pcc = A.addPCC()$
36:             $pcc.addInput(s')$
37:             $o.addInput(pcc.outputStream)$
38:             for all $p \in o.managedPolicies$ do
39:                 $pcc.addPastConditions(pcc.pastConditions)$
40:             end for
41:         end if
42:     end for
43: end procedure
44:
45: procedure addInputsForAtomicConditions(A, o)
46:     for all $s' \in A.S$ do
47:         if $o.requiresAtomicConditionOn(s')$ then
48:             $o.addInput(s')$
49:         end if
50:     end for
51: end procedure
```

each data stream for which *o* has to directly verify an atomic condition.

6 EVALUATION

The objective of our evaluation was twofold:

- Assess the correctness of the enforcement mechanism (see Section 6.1) to check if the outputs produced by the application were consistent with the defined policies.
- Assess the performance overhead introduced by the policy enforcement (see Section 6.2). This, certainly, introduces an overhead, but we have tried to reduce it as much as possible by optimizing the implementation of our operators and by enabling their parallelization to increase scalability.

6.1 Correctness Experiment

As an initial correctness check, we have fed our example application with 10M of transactions and we have checked that

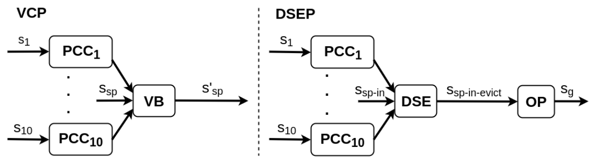

Figure 5: Configurations for the performance experiments.

all observable outputs are correct with respect to the semantics of the enforcement mechanism. To perform this analysis, we have exploited the formal definition of the enforcement semantics (see Section 4.5) and we have used Monpoly [3], a tool for monitoring MFOTL formulae over logs of events. We ran 10 times our DIA example using randomly generated input datasets of 10M transactions and dumping the content of all the data streams produced by the application to properly formatted files (according to the Monpoly input format). We then merged these files into a single one with all the tuples ordered according to their timestamps. Finally, we ran Monpoly over such file, verifying the MFOTL formulae corresponding to $Policy_1$ and $Policy_2$. Monpoly reported no violation in all the 10 executions. Although this does not formally prove the correctness of our implementation, still it provides an empirical evidence of such correctness in the considered example.

6.2 Performance Analysis

The goal of our performance analysis was to assess the ability of our prototype to handle the enforcement of complex privacy policies in the presence of a significantly high number of incoming tuples. More specifically, we assessed the latency, i.e., the average delay after which each tuple exits from the system, and the throughput, i.e., the number of input tuples the DIA processes per second. These are the typical metrics of interest when dealing with DIAs [17].

We considered the two configurations in Figure 5. The configuration on the left hand side refers to the case of VCPs (up to one per data subject) applied on stream s_{sp}, assuming they can contain a variable number of past conditions over streams $s_1 \dots s_{10}$, each consisting of a single integer field. In this case, a ViewBuilder is installed on s_{sp} and a PastConditionChecker on each other stream on which at least one past condition is set. We varied the number of PastConditionCheckers to see how their presence influences the overhead introduced by the policy enforcement.

The configuration of the right handside of Figure 5 refers to the case of DSEPs (up to one per data subject) applied on stream s_g. In this case, we focus on a DataSubjectEvictor operator installed on a subject-specific stream s_{sp-in} that produces data for operator OP, which emits on stream s_g. Also in this case we assume that the conditions checked by the DSE can involve a variable number of past conditions over streams $s_1 \dots s_{10}$ (again, consisting of a single integer field) and, therefore, we have a varying number of PastConditionCheckers as in the previous case.

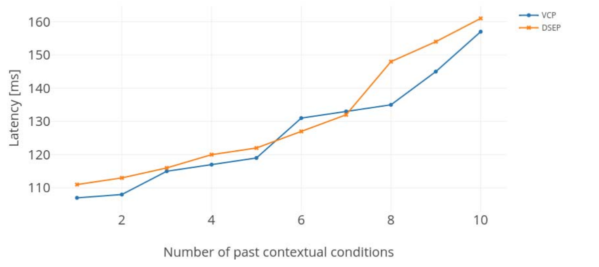

Figure 6: Application latency increase with increasing policy's complexity.

In both configurations, the structure of the input subject-specific stream (i.e. s_{sp} and s_{sp-in}) was *(tupleId: int, dataSubject: string, content: int)*. The *dataSubject* and the *content* fields of each tuple were randomly generated using a uniform distribution. The input subject-specific stream was split into 8 partitions in order to parallelize the processing, as a realistic deployment would need. Flink, in fact, takes care of replicating all operators based on the number of created partitions. In our specific case, the maximum number of replicas was, therefore, 8 for ViewBuilders in the first configuration and DataSubjectEvictors in the second configuration plus 10 PastConditionCheckers, which were not replicated.

We deployed Flink on a cluster of 16 virtual machines, each equipped with 2 CPUs and 8 GB of RAM, so that Flink could allocate the various operators on the available cores without overcoming the maximum available. We made sure to have the machines clocks synchronized. This is necessary in order to have a correct evaluation of temporal properties of distributed systems. Without such clocks synchronization, it may happen that, at some point, a policy results to be violated even if actually it has been not, and vice versa.

In order to fairly measure performance, for each different experiment we proceeded as follows. First, we measured the overall time required to process 200k tuples when they are all submitted in a single burst. In this case, the underlying Flink engine employs a backpressure mechanism that automatically adjusts the input rate at the source based on the current load. Thus, dividing the overall processing time by the number of processed tuples gives us a precise estimate of the maximum input throughput that our application can sustain. Then, we measured the latency for processing individual tuples when they are submitted at 80% of the average maximum input throughput. In all our experiments, we submitted 20k input elements before start measuring, to make sure that the system was in a steady state.

Figure 6 shows the increase of the latency when the number of contextual past conditions in the policy increases, while Figure 7 shows throughput degradation. Both metrics appear to be linearly related to the number of contextual conditions. In the simplest case of a single past condition, the latency increased of 8% when enforcing the VCP and 5% in the case of DSEP, while throughput decreased of around 6% in both the cases. In the worst case in which 10 past conditions on

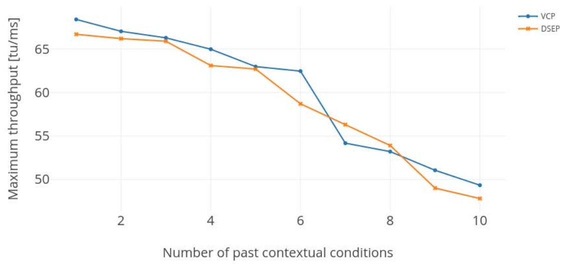

Figure 7: Application throughput degradation with increasing policy's complexity.

10 distinct data streams were defined, which is actually only the case of very complex policies, the latency increased of the 45% when enforcing the VCPs and of the 50% for DSEPs, while throughput decreased of the 28% for VCPs and of 30% for DSEPs. Based on these preliminary observations, we can conclude that the way policies are built has a significant impact on the performance of the enforcement mechanism. According to our initial analysis of existing privacy policies, they usually involve a very limited number of past conditions [2], thus, our approach can work well with this kind of applications. We plan to continue this analysis to confirm this hypothesis.

7 RELATED WORK

There has been a lot of research in the past on how to specify and enforce privacy requirements. Contextual Integrity (CI) [16] is a normative framework for modeling the flow of information between agents and reasoning on communication patterns that cause privacy violations. Barth et al. provide a formalization of CI using a Linear Temporal Logic (LTL), with the goal of precisely specifying privacy laws [2]. Their framework includes privacy relevant concepts like agent role, communication purpose, obligation and focuses on the type of information transmitted. Temporal operators are used to specify conditions on past and future events that must occur in order for a policy to be satisfied. Although the authors clearly mention that the framework can be used by an information processing system to enforce the specified policies, they do not focus on this aspect. Instead, they use the framework in combination with off-line trace-checking techniques to verify that the trace history of agents communication satisfies a given privacy policy.

Ni et al. define a Privacy-Aware Role Base Access Control (P-RBAC) model, which essentially considers various privacy-relevant concepts (purpose, role, obligation) to specify access control policies [15]. With respect to our work, P-RBAC does not target the case of streaming data. Moreover, the model does not foresee any mechanism for controlling the informativeness of accessed data. Carminati et al. propose an access control model for data streams, whose key feature is the ability to model temporal access constraints, like the time window during which access is allowed [7]. Like in our case, access control policies are then implemented through a query rewriting algorithm that exploits a set of secure operators. Nevertheless, there is no explicit focus on privacy aspects. Moreover we believe that data stream access control needs to be re-interpreted in light of modern stream processing. Vigiles [21] is an approach for implementing access control on MapReduce systems, which is implemented in terms of a middleware that applies fine-grained access control rules on the input data of a MapReduce job. Also in this case there is no explicit focus on privacy concepts, nor on streaming data where the interest is on temporal aspects.

Several works have been proposed in the context of adaptive privacy. The problem tacked by Schaub et al. [19] is similar to our, but while they propose a more general conceptual model that they apply on some real scenarios, we instead offer a formal model and target a specific type of platforms. Yang et al. [24] focus instead on social networks, offering an adaptive framework that quantifies users' requirements as a trade-off between their concerns and the incentives of sharing information and maximises the overall utility of sharing.

Recently Yang et al. introduced Jeeves, a programming language for automatically enforcing privacy policies [23] [22]. The purpose of Jeeves is somehow similar to that our approach, as it allows to specify policies that adapt the values of some variables of a program, according to the context in which these are accessed. The context is defined in terms conditions on other variables of the same program. With respect to our work, Jeeves focuses on web applications, rather than dataflow ones. Thus, it does not allow to specify past contextual conditions. Moreover while we rely on the concept of DGH (Section 4.1) to flexibly tune the desired amount of generalization to be applied on a given piece of data, Jeeves only allows to define a single alternative value to be shown for a variable when a given context holds.

8 CONCLUSION

In this paper we proposed a novel framework for specifying, enforcing and checking privacy policies in modern, large-scale DIAs. Our framework models a DIAs as a dataflow and provides (i) a language for specifying privacy policy along with (ii) a dataflow rewriting algorithm to enforce policies within a DIA. We prototyped our solution on top of the Apache Flink and we evaluated its correctness and performance. As a future work, we plan to investigate how our framework can be extended to cope with event time processing and with the case in which multiple conflicting policies, i.e. specified by the same data subject, are enabled at the same time. We also plan to continue the performance assessment of our implementation, for instance by evaluating how performance degrades with increasing number of data subjects or of policies. Finally, we plan to apply our framework to real case studies and privacy regulations, such as the recently introduced *General Data Protection Regulation*.

ACKNOWLEDGMENT

This work is supported by the European Commission grant no. 644869 (H2020, Call 1), DICE.

REFERENCES

[1] Tyler Akidau and others. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment* (2015).

[2] A. Barth, A. Datta, J. C. Mitchell, and H. Nissenbaum. 2006. Privacy and contextual integrity: framework and applications. In *2006 IEEE Symposium on Security and Privacy*. 15 pp.–198. DOI:http://dx.doi.org/10.1109/SP.2006.32

[3] D. Basin, M. Harvan, F. Klaedtke, and E. Zalinescu. 2011. Monitoring Usage-Control Policies in Distributed Systems. In *2011 Eighteenth International Symposium on Temporal Representation and Reasoning*. 88–95. DOI:http://dx.doi.org/10.1109/TIME.2011.14

[4] David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. 2015. Monitoring Metric First-Order Temporal Properties. *J. ACM* 62, 2, Article 15 (May 2015), 45 pages. DOI:http://dx.doi.org/10.1145/2699444

[5] E. Bertino. Data Security and Privacy: Concepts, Approaches, and Research Directions. In *COMPSAC 2016 Proceedings*.

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38.

[7] 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.

[8] C.L. Philip Chen and Chun-Yang Zhang. 2014. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Information Sciences* 275, Supplement C (2014), 314 – 347. DOI:http://dx.doi.org/https://doi.org/10.1016/j.ins.2014.01.015

[9] Min Chen, Shiwen Mao, and Yunhao Liu. 2014. Big Data: A Survey. *Mobile Networks and Applications* 19, 2 (01 Apr 2014), 171–209. DOI:http://dx.doi.org/10.1007/s11036-013-0489-0

[10] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. DOI:http://dx.doi.org/10.1145/1327452.1327492

[11] Michele Guerriero, Saeed Tajfar, Damian A. Tamburri, and Elisabetta Di Nitto. Towards a Model-driven Design Tool for Big Data Architectures. In *BIGDSE '16 Proceedings*.

[12] Doug Henschen. 2012. *2013 Analytics and Info Management Trends*. Technical Report. InformationWeek, http://reports.informationweek.com/abstract/166/9298/.

[13] Yunhan Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Zhuoqing Morley Mao, and Atul Prakash. 2017. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *NDSS*.

[14] Ron Koymans. 1990. Specifying real-time properties with metric temporal logic. *Real-Time Systems* 2, 4 (1990), 255–299. DOI:http://dx.doi.org/10.1007/BF01995674

[15] Q. Ni, E. Bertino, J. Lobo, and S. B. Calo. 2009. Privacy-Aware Role-Based Access Control. *IEEE Security Privacy* 7, 4 (2009), 35–43. DOI:http://dx.doi.org/10.1109/MSP.2009.102

[16] H. Nissenbaum. Privacy as contextual integrity. In *Washington Law Review, 79(1):119-158, 2004*.

[17] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, Berkeley, CA, USA, 293–307. http://dl.acm.org/citation.cfm?id=2789770.2789791

[18] F. Schaub, B. KÃűnings, and M. Weber. 2015. Context-Adaptive Privacy: Leveraging Context Awareness to Support Privacy Decision Making. *IEEE Pervasive Computing* (Jan 2015), 34–43. DOI:http://dx.doi.org/10.1109/MPRV.2015.5

[19] F. Schaub, B. KÃűnings, and M. Weber. 2015. Context-Adaptive Privacy: Leveraging Context Awareness to Support Privacy Decision Making. *IEEE Pervasive Computing* 14, 1 (Jan 2015), 34–43. DOI:http://dx.doi.org/10.1109/MPRV.2015.5

[20] Latanya Sweeney. 2002. Achieving K-anonymity Privacy Protection Using Generalization and Suppression. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.* 10, 5 (Oct. 2002), 571–588.

[21] H. Ulusoy, M. Kantarcioglu, E. Pattuk, and K. Hamlen. Vigiles: Fine-Grained Access Control for MapReduce Systems. In *2014 IEEE International Congress on Big Data*.

[22] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, Dynamic Information Flow for Database-backed Applications. *SIGPLAN Not.* 51, 6 (June 2016), 631–647. DOI:http://dx.doi.org/10.1145/2980983.2908098

[23] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A Language for Automatically Enforcing Privacy Policies. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 85–96. DOI:http://dx.doi.org/10.1145/2103656.2103669

[24] M. Yang, Y. Yu, A. K. Bandara, and B. Nuseibeh. 2014. Adaptive Sharing for Online Social Networks: A Trade-off Between Privacy Risk and Social Benefit. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. 45–52. DOI:http://dx.doi.org/10.1109/TrustCom.2014.10