Precise Analysis of Purpose Limitation in Data Flow Diagrams

Precise Analysis of Purpose Limitation in Data Flow Diagrams

Hanaa Alshareef
Chalmers University of Technology
Gothenburg, Sweden
hanaa@chalmers.se

Katja Tuma
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
k.tuma@vu.nl

Sandro Stucki
Chalmers University of Technology
Gothenburg, Sweden
sandros@chalmers.se

Gerardo Schneider
University of Gothenburg
Gothenburg, Sweden
gerardo@cse.gu.se

Riccardo Scandariato
Hamburg University of Technology
Hamburg, Germany
riccardo.scandariato@tuhh.de

ABSTRACT

Data Flow Diagrams (DFDs) are primarily used for modelling functional properties of a system. In recent work, it was shown that DFDs can be used to also model non-functional properties, such as security and privacy properties, if they are annotated with appropriate security- and privacy-related information. An important privacy principle one may wish to model in this way is *purpose limitation*. But previous work on privacy-aware DFDs (PA-DFDs) considers purpose limitation only superficially, without explaining how the purpose of DFD activators and flows ought to be specified, checked or inferred. In this paper, we define a rigorous formal framework for (1) annotating DFDs with *purpose labels* and *privacy signatures*, (2) checking the consistency of labels and signatures, and (3) inferring labels from signatures. We implement our theoretical framework in a proof-of concept tool consisting of a domain-specific language (DSL) for specifying privacy signatures and algorithms for checking and inferring purpose labels from such signatures. Finally, we evaluate our framework and tool through a case study based on a DFD from the privacy literature.

CCS CONCEPTS

• **Social and professional topics** → **Systems analysis and design**; • **Security and privacy** → **Usability in security and privacy**.

KEYWORDS

Privacy by design, purpose limitation, data flow diagram

ACM Reference Format:
Hanaa Alshareef, Katja Tuma, Sandro Stucki, Gerardo Schneider, and Riccardo Scandariato. 2022. Precise Analysis of Purpose Limitation in Data Flow Diagrams. In *The 17th International Conference on Availability, Reliability and Security (ARES 2022), August 23–26, 2022, Vienna, Austria.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3538969.3539010

ARES 2022, August 23–26, 2022, Vienna, Austria

ACM ISBN 978-1-4503-9670-7/22/08.
https://doi.org/10.1145/3538969.3539010

1 INTRODUCTION

The *European General Data Protection Regulation* (GDPR) entered into force in 2016 after passing European Parliament, and all organizations are required to comply since May 25, 2018. The regulation imposes stringent constraints on the collection and use of individuals' personal data, stipulating heavy penalties in case of violations [11]. Yet, complying with the regulation is challenging for software engineers trying to meet the requirements [18].

Note that privacy does not refer to one particular property but rather to a set of properties, including confidentiality, user consent, the right to be forgotten, purpose limitation, and more. Verifying privacy compliance (even for specific properties) is in general undecidable [19, 21].

In this paper we extend previous work in the area following the *Privacy by Design* (PbD) principle [7], in which any (computerized) personal data processing environment should be designed taking privacy into account from the very beginning of the (software) development process. PbD has been identified as being more tractable than "adding privacy" to already deployed software (or programming only targeting functional properties and add privacy on a later stage of development) [9].

In particular, we are here interested in *data flow diagrams* (DFDs), a graphical representation of how data flows among software components primarily used for modeling functional properties of a system. DFDs can also be used to model non-functional properties, such as security and privacy properties, if they are annotated with appropriate security- and privacy-related information, as shown in [1, 2, 22, 23].

In this paper we enhance DFDs with *purpose limitation*. Antignac et al. [4, 5] proposed an approach to automatically add privacy checks at the design level by means of model transformations and enhancing DFDs with checks for specific privacy concepts. They focused on privacy policies at a very high-level of abstraction. The enhanced diagram is called a *Privacy-Aware Data Flow Diagram* (or PA-DFD for short). In that proposal, the software engineer designs a DFD, pushes a button to obtain a PA-DFD, inspects it manually, and then generates a program template from the PA-DFD that guides the programmer in the concrete implementation of the privacy checks. Antignac et al. describe their transformation from DFDs to PA-DFDs through high-level graphical "rules". Alshareef et al. [2] further extend the approach by providing an algorithm for the transformations and a reference implementation.

In this paper, we extend the above mentioned work and define a rigorous formal framework, algorithms and a proof-of-concept implementation for modeling purpose limitation in DFDs.

In more detail, our contributions are:

(1) In §3, we define a rigorous mathematical framework for (i) annotating DFDs with *purpose labels* and *privacy signatures*, (ii) checking the consistency of labels and signatures, and (iii) inferring labels from signatures.
(2) In §4.3, we describe a proof-of-concept tool implementing features (i–iii) of our theoretical framework. Concretely, we design and implement a domain-specific language (DSL) to be used by domain-experts for specifying privacy signatures. These signatures can then be used by designers to annotate DFDs (i). We also describe and implement algorithms to automatically check (ii) and infer (iii) purpose labels consistent with such annotated DFDs.
(3) In §5, we evaluate our framework and tool through a case study based on a DFD from the privacy literature.

We cover relevant background information in §2, discuss related work in §6 and draw conclusions in §7.

2 PRELIMINARIES

2.1 GDPR

Regulation (EU) 2016/679, better known as the European *General Data Protection Regulation* (GDPR), stipulates rules concerning "the protection of natural persons with regard to the processing of personal data and rules relating to the free movement of personal data", and it "protects fundamental rights and freedoms of natural persons and in particular their right to the protection of personal data" [11]. It contains 99 articles regulating *personal data* processing, and it is organized around a number of key concepts, most notably its seven *principles* of personal data processing. Relevant to this paper is the principle of *purpose limitation*.

According to Article 5 of the regulation, "Personal data shall be," among other things, "collected for specified, explicit and legitimate purposes and not further processed in a manner that is incompatible with those purposes; further processing for archiving purposes in the public interest, scientific or historical research purposes or statistical purposes shall, in accordance with Article 89(1), not be considered to be incompatible with the initial purposes ('purpose limitation')[.]"

In order to check, and eventually enforce, that data assets are used for their intended purpose in a computer system, one must track the flow of both the data as well as its associated purpose through the system.

2.2 Data Flow Diagrams

A *data flow diagram* (DFD) is a graphical representation used to model the flow of information among software components. Based on the "data flow graph" computation models, DFDs were popularized in the late 1970s, starting with the software engineering trend on structured design.

Fig. 1 shows two simple examples of DFDs (we will handle more complex cases later). The terminology we use is as follows: the diagrams are composed of *activators* and *flows*. Activators can be *external entities* (rectangles, representing for instance end users and third party systems), *processes* (circles, computation applied to the data in the system) and *data stores* (double-lined entities).

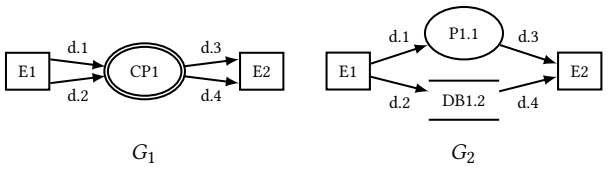


Figure 1: Example DFDs

Processes may represent detailed low-level operations or be high-level, representing complex functionalities that could be refined into more detailed processes. *Composite* processes are represented by a double-lined circle or ellipse. The flow of data is represented by *data flow* arrows.

As already mentioned, though DFDs have originally introduced for functional properties, some extensions have been done in order to incorporate privacy (and security) concepts. For that, Antignac et al [4, 5] extended the standard notation of DFDs with *data deletion* type of flow, to indicate specific piece of data is to be deleted from a database. This extension is referred to as *Business-oriented DFD* (B-DFD).

2.3 DFDs as Graphs

Following our previous work on PA-DFDs [1, 2], we formally represent DFDs as *attributed multigraphs* with activators as nodes and flows as edges. For the benefit of the reader, we reproduce the relevant definitions in this section. In the following, we use "harpoon" arrows ($\rightharpoonup$) to denote partial maps.

Definition 2.1. An *attributed multigraph* (or simply *graph*) G is a tuple $G = (\mathcal{N}, \mathcal{F}, \mathcal{A}, \mathcal{V}, s, t, \ell_{\mathcal{N}}, \ell_{\mathcal{F}})$ where $\mathcal{N}$, $\mathcal{F}$, $\mathcal{A}$ and $\mathcal{V}$ are sets of nodes, edges, attributes and attribute values, respectively; $s, t\colon \mathcal{F} \to \mathcal{N}$ are the source and target maps; $\ell_{\mathcal{N}}\colon \mathcal{N} \to (\mathcal{A} \rightharpoonup \mathcal{V})$ and $\ell_{\mathcal{F}}\colon \mathcal{F} \to (\mathcal{A} \rightharpoonup \mathcal{V})$ are attribute maps that assign values for the different attributes to nodes and flows, respectively.

Examples of attributed multigraphs are shown in Fig. 1. The graph G_1 has nodes $\mathcal{N} = \{\text{E1}, \text{E2}, \text{CP1}\}$ and edges $\mathcal{F} = \{\text{d.1}, \ldots, \text{d.4}\}$. G_1 is a multigraph since both edges d.1 and d.2 connect the same source and target nodes: $s(\text{d.1}) = s(\text{d.2}) = \text{E1}$ and $t(\text{d.1}) = t(\text{d.2}) = \text{CP1}$. Attributes allow us to specify properties of activators and flows, such as their type or associated privacy information.

For example, the graph G_1 has two kinds of nodes, external entities and composite processes. We formalize this by defining its attribute and value sets as $\mathcal{A} = \{\text{type}\}$ and $\mathcal{V} = \{\text{ext}, \text{cproc}\}$, and its node attribute map as $\ell_{\mathcal{N}}(\text{E1})(\text{type}) = \ell_{\mathcal{N}}(\text{E2})(\text{type}) = \text{ext}$ and $\ell_{\mathcal{N}}(\text{CP1})(\text{type}) = \text{cproc}$. Note that the attribute maps are partial, i.e. nodes and edges may lack values for certain attributes. If we extend the value set $\mathcal{V}$ with types for processes (proc) and data stores (db), we can encode the graph G_2 shown in Fig. 1 similarly.

Henceforth, we use the letters n, m to denote nodes and e, f to denote edges. We write $e\colon n \rightsquigarrow m$ to indicate that e has source $s(e) = n$ and target $t(e) = m$. For example, we have d.1: E1 $\rightsquigarrow$ CP1 in G_1. We use "." to select attributes, writing $n.a$ for $\ell_{\mathcal{N}}(n)(a)$ and $f.a$

for $\ell_{\mathcal{F}}(f)(a)$. For example, E1.type = ext in G_1. The set $S(G) \subseteq \mathcal{N}$ of *source nodes* in G is defined as $S(G) = \{n \mid \exists e.s(e) = n\}$; similarly, $T(G)$ denotes the set of *target nodes* in G. The set $\text{in}(n)$ of *input flows* of a node n is defined as $\text{in}(n) = \{f \mid t(f) = n\}$, and similarly for the set of *output flows* $\text{out}(n)$.

DFDs. A DFD is an attributed multigraph with a fixed choice of attributes $\mathcal{A} = \{\text{type}\}$ and values $\mathcal{V} = \mathcal{T}_{\text{dn}} \uplus \mathcal{T}_{\text{df}}$. The sets of *data node types* $\mathcal{T}_{\text{dn}}$ and *data flow types* $\mathcal{T}_{\text{df}}$ are defined as

$$\mathcal{T}_{\text{dn}} = \{\text{ext}, \text{proc}, \text{db}, \text{cproc}\} \qquad \mathcal{T}_{\text{df}} = \{\text{pf}, \text{df}\}.$$

We adopt Antignac et al.'s notion of *business-oriented DFDs (B-DFDs)* [4] and distinguish between two types of flows: *plain flows* (pf) and *deletion flows* (df).[1]

Since the type attribute plays an important role in B-DFDs, we introduce shorthands for typing activators and flows. We write $n : t$ to abbreviate $n.\text{type} = t$, and $f : n \rightsquigarrow_t m$ to indicate that $f : n \rightsquigarrow m$ and $f.\text{type} = t$.

We require that DFDs be *well-formed*. We adopt the standard rules from the DFD literature for well-formed DFDs: diagrams should not contain loops (flows with identical source and target activators), activators cannot be isolated (disconnected from all other activators), and processes must have at least one incoming and outgoing flow (see e.g. [10, 12]).

Definition 2.2. A *B-DFD* (or simply *DFD*) is an attributed multigraph G, where $\mathcal{A}_G = \{\text{type}\}$ and $\mathcal{V}_G = \mathcal{T}_{\text{dn}} \uplus \mathcal{T}_{\text{df}}$. In addition, for all flows f and activators n, m,

- $n.\text{type} \in \mathcal{T}_{\text{dn}}$ and $f.\text{type} \in \mathcal{T}_{\text{df}}$;
- if $f : n \rightsquigarrow_{\text{df}} m$ then $m : \text{db}$;
- if $f : n \rightsquigarrow m$ then $n \neq m$;
- if $n : \text{cproc}$ or $n : \text{proc}$ then $n \in S(G)$ and $n \in T(G)$
- if $n : \text{ext}$ or $n : \text{db}$ then $n \in S(G)$ or $n \in T(G)$

For more details about DFDs, see [1].

3 MODELING PURPOSE IN DFDS

As a first step towards our goal of modeling privacy properties, and in particular purpose limitation, in DFDs, we define a rigorous mathematical framework for annotating DFDs with *purpose labels* and *privacy signatures*. This framework will serve as the theoretical basis for developing the concrete DSL and algorithms presented in §4.3 as well as our proof-of-concept tool. The framework is graph-theoretic and builds on the definitions of DFDs and their refinements recapitulated in §2.3.

To define our theoretical framework, we must first identify a formal notion to represent the informal concept of *purpose* used in the GDPR. Recall, from §2.1, that the GDPR mandates that personal data shall be collected and process only for "specified, explicit and legitimate purposes" [11]. This is the principle of purpose limitation. Let us concede immediately that purpose limitation and the notion of "purpose" itself are legal concepts, and that it is not possible, in general, to express such concepts in a fully formal way. The best we can hope for is to formalize a useful approximation.

[1] Alshareef et al. [1, 2], use a more fine-grained notion of flow types, distinguishing flows based on their source and target activators. There is no need for such detailed flow typing in the present paper, but our theoretical framework readily generalizes to Alshareef et al.'s definitions.

With this caveat in mind, our goal is to map the legal concepts underlying the principle of purpose limitation to corresponding primitives in the graphical language of DFDs, and to extend that language with new primitives where necessary. It is very natural to model data via flows, and data processing and collection via activators (such as processes and data stores), but there is no obvious counterpart to the concept of "purpose" in the language of DFDs. To address this shortcoming, we introduce two additional notions in our graph-theoretic treatment of DFDs: *purpose labels* on data flows to represent the intended purpose for which a piece of data is to be used, and *privacy signatures* on activators to model the impact of processing and storage on these purpose labels.

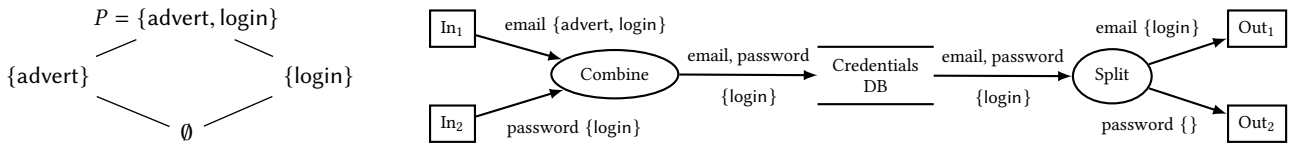
3.1 Purpose Labels

Since DFDs are a very general modeling tool with applications in many contexts, we wish to keep our formal notion of *purpose* equally general and widely applicable. We take the view that there are many possible ways to represent purpose mathematically, and that the most appropriate choice depends on the domain and context of a given model.

We therefore assume that we are given, for each DFD G, a pre-ordered set $(\mathcal{P}_G, \sqsubseteq_G)$ of *purpose labels* (or simply *purposes*) that is used to annotate the flows of G. To avoid clutter, we will usually omit the superscripts and just write $\mathcal{P}$ and $\sqsubseteq$. The concrete choice of $\mathcal{P}$ and $\sqsubseteq$ depends on the context in which G is used and should be made by a domain-expert.

The elements of $\mathcal{P}$ are denoted by the letters $p, q, \ldots$ and represent the intended purpose for which a given piece of data may be used. The preorder $\sqsubseteq$ on purpose labels models *subsumption*: if $p \sqsubseteq q$, we say that p is a *sub-purpose* of q or that q *subsumes* p. Intuitively, $p \sqsubseteq q$ means that p is a more specific purpose than q, i.e. a q-labeled piece of data can also be used for the more specific purpose p. For example, a DFD modeling parts of an e-commerce system may be annotated with purpose labels $\mathsf{ProductSelection}, \mathsf{Payment}, \mathsf{Purchase} \in \mathcal{P}$, where $\mathsf{ProductSelection} \sqsubseteq \mathsf{Purchase}$ and $\mathsf{Payment} \sqsubseteq \mathsf{Purchase}$ because a purchase may involve both product selection and payment, but $\mathsf{ProductSelection} \not\sqsubseteq \mathsf{Payment}$ because product selection and payment are independent activities. Hence, a piece of data (such as a credit card number) could be used for all three purposes if labeled Purchase, but only for payment if labeled Payment.

In the remainder of this paper, we will often assume that $\mathcal{P}$ is the *powerset* $\mathcal{P} = 2^P$ of a finite set of *atomic* or *basic purposes* P, and that purposes are ordered by *set inclusion* $\sqsubseteq = \subseteq$. Intuitively, atomic purposes $a, b, \ldots \in P$ represent the basic functions a system can perform. A purpose label $p \in \mathcal{P}$ is then a collection of all the atomic purposes that a given piece of data may be used for: if $a \in p$, then the data can be used for the atomic purpose a, if $a \notin p$, it cannot. A purpose label q subsumes another purpose label p if $p \subseteq q$. This corresponds to the intuition that it is safe to use a q-labeled piece of data for the more restricted set $p \subseteq q$ of basic functions. Fig. 2 shows an example for the set of atomic purposes $P = \{\text{advert}, \text{login}\}$. The example models a toy system that can perform only two basic functions: advertising (advert) and user log-in (login). The Hasse diagram in Fig. 2a shows how purpose labels are ordered. At the top of the diagram is the set P itself: data



(a) The powerset lattice of $P = \{\text{advert}, \text{login}\}$. **(b) A DFD combining and splitting data with purpose labels drawn from $\mathcal{P} = 2^P$.**

Figure 2: A simple purpose lattice $\mathcal{P} = 2^P$ and a $\mathcal{P}$-annotated DFD.

labeled P may be used for any purpose whatsoever; at the bottom of the diagram is the empty set $\emptyset$: data labeled $\emptyset$ may not be used for any purpose, i.e. it cannot be used at all. The two singleton sets $\{\text{advert}\}$ and $\{\text{login}\}$ are used to label data that can be used for exactly one of the basic purposes.

In general, the powerset 2^P of a finite set P forms a finite bounded lattice with maximum P and minimum $\emptyset$, and with set intersection $\cap$ as the greatest lower bound (GLB) and set union $\cup$ as the least upper bound (LUB). We have already noted that the extremal labels P and $\emptyset$ are useful to minimally or maximally constrain the use of data. The lattice operations $\cap$ and $\cup$, on the other hand, are useful when determining the purpose labels of outputs and inputs of processes that combine and split data, respectively. Fig. 2b shows an example. The "Combine" process simply takes its inputs "email" and "password" and combines them into a tuple (to be stored in a credentials database). Initially, the "email" and "password" flows are labeled $p_1 = \{\text{advert}, \text{login}\} = P$ and $p_2 = \{\text{login}\}$, indicating that the user's email address can be used for any purpose, while their password can only be used for log-in. What purposes may we use the combined data for? Because the tuple "email, password" contains both data, we must take care to only use it for purposes covered by *both* p_1 and p_2, i.e. the atomic purposes in $p_1 \cap p_2 = \{\text{login}\}$. Conversely, we may ask what purpose label we should use for the input to the "Split" process in the figure, if given only the labels $q_1 = \{\text{login}\}$ and $q_2 = \emptyset$ of its two outputs. Since the label of the input "email, password" must cover the use of *either* component separately, the appropriate label is $q_1 \cup q_2 = \{\text{login}\}$.

Although we only explicitly cover the case where $\mathcal{P}$ is a powerset lattice here, the same reasoning extends immediately to any domain where purpose labels can be represented as a bounded lattice. We will revisit this idea in §3.2.2, where we define a language for privacy signatures based on lattice expressions.

3.2 Signatures: Propagating Purpose Labels

The discussion at the end of the previous section about the use of the lattice operations for determining the correct purpose labels on the inputs and outputs of processes illustrates a more general point: as data flows through a DFD and is changed by different activators, the purpose of the data changes as well, and the purpose labels of the corresponding flows must be updated. Processes that simply split or combine data are easy to reason about because they do not actually change the underlying data. Hence it is straightforward to relate purpose labels of their inputs and outputs. But there are no general rules for doing this when processes perform arbitrary computations or when data is read from or written to external entities or data stores. In such cases, a designer has to explicitly model the effect of an activator on the purpose labels of its inputs and outputs.

In theory, almost any type of relationship between purpose labels of inputs and outputs are imaginable. Indeed, a process might – in theory – simply forget all its inputs and produce random outputs. In this extreme case, the purpose labels on the inputs and outputs of the process need not be related at all.

In practice, however, we argue that, for any given application domain, there ought be a finite number of useful combinations of input and output labels, corresponding to typical operations performed by systems in that domain. For example, a process might *aggregate* data from two inputs with purpose labels p_1 and p_2, in which case the label of the aggregated output should be $p_2 \cap p_2$, just as for the "Combine" process in the previous example – unsurprisingly, as pairing is a particular type of aggregation.

As an other example, a process might *anonymize* one of its inputs, in which case the anonymized output may be given any purpose label whatsoever, provided that the anonymization is done properly (i.e. any trace of personal information has been removed). We call the combinations of input and output purposes associated with such generic (but domain-specific) operations *privacy signatures*.

Before we give our formal definition of privacy signatures, we need to introduce a bit of auxiliary notation. Let I be a finite set, e.g. $I = \{1, 2, \ldots, n\}$ for some natural number n. We think of finite maps $\mathbf{p} \in \mathcal{P}^I$ as *tuples* or *vectors* $\mathbf{p} = (p_1, p_2, \ldots, p_n)$ of purpose labels with *index set* I. We use $\mathbf{p}(i)$ and p_i interchangeably to denote the i-th component of $\mathbf{p}$. The subsumption order $\sqsubseteq$ extends pointwise to vectors: we write $\mathbf{p} \sqsubseteq \mathbf{q}$ if $p_i \sqsubseteq q_i$ for all $i \in I$. In the following, we will use purpose vectors to represent collections of purpose labels indexed by flows ($I \subseteq \mathcal{F}$), activators ($I \subseteq \mathcal{N}$) or natural numbers ($I = \{1, 2, \ldots, n\}$).

We take the view that privacy signatures are fully characterized by how they act on purpose labels, i.e. if two privacy signatures combine input and output labels in the same way, they are indistinguishable. Formally, we therefore model privacy signatures simply as functions from (vectors of) input purpose labels to (vectors of) output purpose labels.

Definition 3.1. Let I and O be finite sets of input and output names, respectively. A *privacy signature* or *purpose propagation function (PPF)* is a monotone map $\sigma : \mathcal{P}^I \to \mathcal{P}^O$. That is, for all $\mathbf{p}, \mathbf{q} \in \mathcal{P}^I$, if $\mathbf{p} \sqsubseteq \mathbf{q}$, then $\sigma(\mathbf{p}) \sqsubseteq \sigma(\mathbf{q})$.

A signature $\sigma : \mathcal{P}^I \to \mathcal{P}^O$ thus computes labels on outputs from labels in inputs or, in other words, σ *propagates* purpose labels on

inputs to purpose labels on outputs – whence the name purpose propagation function.

For example, consider the privacy signature σ_{agg} associated with the Combine activator in Fig. 2b or any other process *aggregating* two inputs into one output. There are two inputs and one output, so we choose $I = \{1, 2\}$ and $O = \{1\}$. This means that σ_{agg} must map pairs of input purposes (p_1, p_2) to a single output purpose $\sigma_{\text{agg}}(p_1, p_2)$. As explained earlier, an appropriate output purpose for aggregation is the GLB of p_1 and p_2, i.e. $\sigma_{\text{agg}}(p_1, p_2) = p_1 \cap p_2$.

We use the terms "privacy signature" and "purpose propagation function" interchangeably. The former explains *what* PPFs represent, while the latter describes *how* they work. The term *propagation function* was coined by Tuma et al. [22] who uses it to model the effect of activators on the *confidentiality* of inputs and outputs; instead of purpose labels, their functions propagate *security labels* from inputs to outputs.

The monotonicity requirement says that, if we restrict the purpose label on any input, the purpose labels on the outputs should not become more general as a consequence. This captures the intuition that, if a data subject limits her consent to a more restrictive set of purposes, the system should be more constrained in its use of the data both before *and after* processing the data. Consider again the privacy signature σ_{agg} representing aggregation: monotonicity holds because, whenever $p_1 \subseteq p_1'$ and $p_2 \subseteq p_2'$, we have $p_1 \cap p_2 \subseteq p_1' \cap p_2'$, and hence $\sigma_{\text{agg}}(p_1, p_2) \subseteq \sigma_{\text{agg}}(p_1', p_2')$.

Given a set of purpose labels $\mathcal{P}$, we denote by $\text{Sig}(\mathcal{P})$ the set of possible signatures ranging over $\mathcal{P}$. That is,

$$\text{Sig}(\mathcal{P}) = \{\sigma \colon \mathcal{P}^I \to \mathcal{P}^O \mid I, O \text{ finite } \}.$$

3.2.1 Purpose-Annotated DFDs. With the definitions of purpose labels and privacy signatures in place, it is now easy to extend our earlier definition of DFDs, Def. 2.2, to incorporate purpose annotations. We simply equip DFDs with an additional attribute purpose that denotes the purpose label and privacy signature of the corresponding flows and activators, respectively. To this end, we also extend the set of possible values $\mathcal{V}_G$ with the set of purpose labels $\mathcal{P}$ and signatures $\text{Sig}(\mathcal{P})$.

Definition 3.2. A *purpose-annotated DFD* (or simply *annotated DFD*) is a DFD G with extended attribute set $\mathcal{A}_G = \{\text{type}, \text{purpose}\}$ and extended value set $\mathcal{V}_G = \mathcal{T}_{\text{dn}} \uplus \mathcal{T}_{\text{df}} \uplus \mathcal{P} \uplus \text{Sig}(\mathcal{P})$, such that $f.\text{purpose} \in \mathcal{P}$ and $n.\text{purpose} \in \mathcal{P}^{\text{in}(n)} \to \mathcal{P}^{\text{out}(n)}$ for all activators n and flows f.

Note that, for simplicity, we assume the input and output names of a signature $n.\text{purpose}$ to be exactly the set of input and output flows $\text{in}(n)$ and $\text{out}(n)$ of n. In practice, this requirement can be relaxed: it suffices to establish a one-to-one correspondence between the input/output flows of activators and the input/output names of their signatures.

In the remainder of the paper, we use the following shorthands when working with purpose annotations. We write p_f for $f.\text{purpose}$ and σ_n for $n.\text{purpose}$. Given a set $F \subseteq \mathcal{F}$ of flows, we define the vector $\mathbf{p}_F$ as $\mathbf{p}_F(f) = p_f$. For example, $\mathbf{p}_{\text{in}(n)}$ denotes the vector of input purpose labels for a given activator n. Because every flow f has exactly one source activator $n = s(f)$, and every activator n has exactly one signature σ_n, there is canonical map $\sigma_f : \mathcal{P}^{\text{in}(s(f))} \to \mathcal{P}$ defined as $\sigma_f(\mathbf{p}) = \sigma_{s(f)}(\mathbf{p})(f)$ that propagates input purposes of $s(f)$ to f. We call this map σ_f the *signature* of f. The slight abuse of terminology is justified by the fact that the signature σ_n of an activator n is the pairing of the signatures of its outputs, i.e. $\sigma_n(\mathbf{p})(f) = \sigma_f(\mathbf{p})$ for $f \in \text{out}(n)$.

Def. 3.2 ensures that every flow in an annotated DFD has a purpose and that every activator has a signature. But it does not guarantee that the purpose labels in the input and output flows of an activator n are *consistent* with the signature of n. For example, we could annotate the Combine activator in Fig. 2b with the bogus signature σ defined as $\sigma(\_, \_) = \emptyset$, i.e. σ returns the constant output purpose $\emptyset$ irrespective of the input purposes. The output label returned by the signature σ is clearly inconsistent with the actual purpose label on the output flow of Combine, which is $\{\text{login}\}$. Surely, an output that has purpose label $\emptyset$, and therefore cannot be used for any purpose, should not be passed on to a downstream activator for the purpose of log-in! However, had σ been defined as $\sigma(\_, \_) = P$, there would not have been a problem: the output of Combine could then have been used for any purpose, including log-in. More generally, we require that the purpose labels computed by signatures subsume (i.e. are more permissive than) those specified via annotations.

Definition 3.3. An annotated DFD G is *consistent* if, for all flows $f : n \rightsquigarrow m$, we have $p_f \sqsubseteq \sigma_f(\mathbf{p}_{\text{in}(n)})$.

3.2.2 A Language for Signatures. So far, we have deliberately kept the representation of purposes and signatures abstract. But, if we wish to use our theoretical framework to model and reason about concrete systems, we must choose concrete representations for purposes and signatures that can be understood and manipulated by human beings (designers, domain experts) and software tools (such as those described in §4.3). Earlier in this section, we have discussed a possible representation for purpose labels that fits this requirement: a domain expert may fix a finite set P of atomic purpose labels – typically a short list of strings such as $P = \{\text{"login"}, \text{"advert"}\}$ – that represent the basic functions of systems in the given domain. Purpose labels are then simply subsets of P, such as $\{\}$, $\{\text{"login"}\}$, etc. In other words, purpose labels are elements of the powerset lattice $\mathcal{P} = 2^P$. We call this the *finite set* representation. The finite set representation is both intuitive for human beings – the purpose of a data asset is just the list of basic functions it may be used for – and easy to implement algorithmically.

Assuming a finite set representation for purpose labels, signatures are monotone maps between vectors of finite sets. It may be tempting to simply represent such maps directly, e.g. as tables or hash-maps. But such a representation quickly becomes intractable, even when the number of atomic purposes $|P|$ is small. For example, consider the set P from Fig. 2 which contains only $|P| = 2$ atomic purposes. A signature for a process activator with just two inputs, such as the aggregation signature σ_{agg} of the Combine activator in Fig. 2b, would require a table with $(2^{|P|})^2 = 16$ entries. If we double the number of atomic purposes to $|P| = 4$, then the number of table entries for the same signature grows to 256, even though the semantics of the signature remains the same: it simply computes the intersection of the two input sets. Clearly, a direct representation of signatures as finite maps is inefficient both for human beings and for algorithms.

We therefore choose an alternative representation that is more intuitive to work with and also more compact: a small domain-specific language (DSL) of *lattice expressions*. The grammar for lattice expressions e is

$$e ::= x \mid p \mid \top \mid \bot \mid e \sqcap e \mid e \sqcup e$$

where x denotes a variable name drawn from a finite set V (typically the set of input names I of some signature) and $p \in \mathcal{P}$ denotes a constant. The remaining expression formers correspond to the usual operators of a bounded lattice. The top ($\top$) and bottom ($\bot$) constants are strictly speaking redundant (they are subsumed by constant expressions p) but turn out to be useful in practice. Using lattice expressions, the aggregation signature σ_{agg} of the Combine activator from Fig. 2b can be expressed compactly as "$x \sqcap y$", where x and y are the names of the two inputs labels.

Let us assume, once again, that $\mathcal{P} = 2^P$ is the powerset lattice generated by a finite set of atomic purposes P. Then, given an *environment* $\rho : V \to \mathcal{P}$, i.e. a map that assigns concrete purpose labels to variables, we can *evaluate* an expression e to a purpose $[\![e]\!]_\rho \in \mathcal{P}$ in the obvious way.

$$[\![x]\!]_\rho = \rho(x) \qquad [\![\top]\!]_\rho = P \qquad [\![e_1 \sqcap e_2]\!]_\rho = [\![e_1]\!]_\rho \cap [\![e_2]\!]_\rho$$
$$[\![p]\!]_\rho = p \qquad [\![\bot]\!]_\rho = \emptyset \qquad [\![e_1 \sqcup e_2]\!]_\rho = [\![e_1]\!]_\rho \cup [\![e_2]\!]_\rho$$

Note that the definition of $[\![-]\!]$ immediately generalizes to arbitrary bounded lattices (other than finite powersets).

Lattice expressions evaluate to monotone functions, which makes them a suitable DSL for specifying signatures.

Lemma 3.4 (monotonicity). *The function $[\![e]\!] : \mathcal{P}^V \to \mathcal{P}$ is monotone for any lattice expression e.*

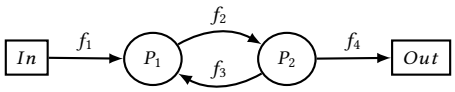
Proof. By straightforward induction on the structure of e and monotonicity of the lattice operations. (See App. A for details.) □

As discussed in §3.2.1, we may specify the signature σ_n of an activator by defining a signature σ_f for each of its output flows $f \in \text{out}(n)$. Thus, given finite sets I and O of input and output names, respectively, a *syntactic signature* $\mathbf{e}$ is a tuple $\mathbf{e} = (e_j)_{j \in O}$ of $|O|$ lattice expressions – one for each output – with variables in $V = I$. We extend evaluation pointwise to syntactic signatures, i.e. we define $[\![\mathbf{e}]\!] : \mathcal{P}^I \to \mathcal{P}^O$ as $[\![\mathbf{e}]\!]_{\mathbf{p}}(j) = [\![e_j]\!]_{\mathbf{p}}$. Finally, we define *syntactically annotated DFDs* in the same way as annotated DFDs, but with the purpose attributes of activators n ranging over syntactic signatures $\mathbf{e}_n$ instead of signatures σ_n.

For simplicity, we will assume that a syntactic signature $\mathbf{e}_n$ of an activator n has input set $I = \text{in}(n)$ and output set $O = \text{out}(n)$, though, in practice, we merely require that there is a one-to-one correspondence between the input/output names of $\mathbf{e}_n$ and the input/output flows of n. In fact, we believe that there is generally only a small number of useful (syntactic) signatures for any given application domain, and that it is the task of domain experts to define, ahead of time, a relevant collection of (syntactic) signatures to be used by software designers working in that application domain. By its nature, this collection of signatures will not be specific to any given DFD, and hence the names of inputs and outputs of those signatures will not, in general, match those of the flows in a particular DFD. Instead, a software designer will have to specify a correspondence between the two when annotating a DFD. This is the approach taken in our proof-of-concept tool, which we describe in more detail in §4.3.

3.2.3 A Sketch of Purpose Inference. A consistently annotated DFD contains some redundant information. The consistency requirement means that designers cannot annotate flows and activators with arbitrary, incompatible purpose labels and signatures. Instead, the signature on a given node sets an upper bound on the purpose labels of its outgoing flows. And, since every flow is an output of some node (namely its source), the signatures in an annotated DFD effectively constrain the purpose labels on all its flows. Indeed, one may wonder whether the purpose labels on flows are entirely redundant, in the sense that they could be be *inferred* from the signatures.

The alternative name we use for privacy signatures – *purpose propagation functions* (PPFs) – suggests a naive approach for inferring purpose labels: simply propagate purpose labels through the DFD, one activator at a time, starting with the input activators (external entities with only output flows). The problem with this method is that it cannot deal with *cycles* in the DFD. Consider, for instance, the following DFD.



Assume each activator n carries a signature σ_n. It is easy to derive a purpose label for the flow f_1: because the *In* activator only has output flows (but no inputs) the signature $\sigma_{In} : \mathcal{P}^\emptyset \to \mathcal{P}^{\{f_1\}}$ is a constant function that assigns a fixed purpose $p_1 = \sigma_{In}()(f_1)$ to f_1. But things get complicated for the next activator, P_1, which has two input flows, f_1 and f_3. To apply σ_{P_1}, we need to compute their purpose labels p_1 and p_3. We have a value for p_1 but we have not yet computed p_3. Indeed, the value of p_3 depends on that of p_2, via σ_{P_2}, which depends on p_3 itself – we are stuck in a cycle.

A possible workaround for this problem is to ask designers to specify more details about the system that is being modeled, e.g. a sequence of events (a trace) that witnesses the order in which activators are to be executed [see e.g. 22]. Here, we adopt a different approach, borrowing ideas from program analysis, in particular from *abstract interpretation* and *data flow analysis*.[2] The advantage of this approach is that it does not require any additional information from the designer.

Let us recapitulate some basic facts from order theory that are the heart of our data flow analysis. It is well known that, if $\mathcal{L}$ is a finite bounded lattice and I is a finite set, then the set $\mathcal{L}^I$ of I-indexed vectors over $\mathcal{L}$ is also a finite bounded lattice. The lattice order and operations are extended pointwise, e.g. $\boldsymbol{\bot} = (\bot, \ldots, \bot)$ and $\mathbf{p} \sqcap \mathbf{q} = (p_1 \sqcap q_1, \ldots, p_n \sqcap q_n)$. Another well-know fact about finite bounded lattices $\mathcal{L}$ is that any monotone function $F : \mathcal{L} \to \mathcal{L}$ has a *least fixpoint* $\mu(F) \in \mathcal{L}$. Recall that an element $x \in \mathcal{L}$ is a *fixpoint* of F if $F(x) = x$, and that x is a *least* fixpoint if $x \sqsubseteq y$ for any other $y \in \mathcal{L}$ such that $F(y) = y$. Furthermore, this least fixpoint can be computed as the limit of the (necessarily finite) chain $\bot \sqsubseteq F(\bot) \sqsubseteq F(F(\bot)) \sqsubseteq \cdots \sqsubseteq \mu(F)$, where $\bot$ is the minimum of $\mathcal{L}$. The main idea behind data flow analysis is to infer a static property of a system

[2] Despite the name, *data flow analysis* has nothing to do with *data flow diagrams*.

(represented by elements of a lattice $\mathcal{L}$) by iteratively computing better and better approximations of the desired property (via a monotone function F) until a fixpoint is reached. By re-formulating the problem of purpose inference as a data flow analysis, we can leverage these properties and compute a consistent annotation for the flows of a DFD as the least fixpoint of some well-chosen monotone function.

Given a DFD G (not necessarily annotated), we define a *flow annotation* for G as a map $\mathbf{p} \in \mathcal{P}^{\mathcal{F}_G}$ from flows to purpose labels or, equivalently, a vector of purpose labels indexed by $\mathcal{F}_G$. Because the set $\mathcal{F}_G$ is finite, a flow annotation $\mathbf{p}$ is an element of the finite bounded lattice $\mathcal{P}^{\mathcal{F}_G}$. Given a flow annotation $\mathbf{p}$ and a subset $F \subseteq \mathcal{F}_G$ of flows, we write $\mathbf{p}|_F$ for the *restriction* of $\mathbf{p}$ to F, i.e. the vector $\mathbf{p}|_F \in \mathcal{P}^F$ where we simply forgot the purpose labels of flows that are not in F. Assume now that we are given a signature σ_n for each activator n in G. As explained earlier, this is equivalent to specifying a signature σ_f for each flow $f \in \mathcal{F}_G$. It is easy to extend each such signature σ_f into a function $\Sigma_f : \mathcal{P}^{\mathcal{F}_G} \to \mathcal{P}$, defined as $\Sigma_f(\mathbf{p}) = \sigma_f(\mathbf{p}|_{\text{in}(s(f))})$ which takes a full flow annotation as its input and ignores all those purpose labels that are not relevant for f. Combining the various Σ_f, we obtain a single function $\Sigma_G : \mathcal{P}^{\mathcal{F}_G} \to \mathcal{P}^{\mathcal{F}_G}$ defined as $\Sigma_G(\mathbf{p})(f) = \Sigma_f(\mathbf{p})$. We call Σ_G the *global purpose propagation function (GPFF)* of G. Intuitively, Σ_G takes a full flow annotation for G as its input and propagates all the purpose labels on all the input flows in G to all the output flows in G, simultaneously. Here, we derived Σ_G from individual signatures σ_n, but it is an easy exercise to do the opposite, i.e. derive individual signatures (for activators or flows) from a given map $\Sigma_G : \mathcal{P}^{\mathcal{F}_G} \to \mathcal{P}^{\mathcal{F}_G}$. Hence, it does not matter if we specify PFFs per-activator, per-flow, or globally – all three are equally expressive.

We can restate the *consistency* condition form Def. 3.3 in terms of the GPFF as follows: an annotated DFD G is consistent if $\mathbf{p}_{\mathcal{F}_G} \sqsubseteq \Sigma_G(\mathbf{p}_{\mathcal{F}_G})$. Clearly, any flow annotation $\mathbf{p}$ that is a fixpoint of Σ_G gives rise to a consistently annotated DFD since the above consistency condition immediately follows from $\mathbf{p} = \Sigma_G(\mathbf{p})$. Because signatures are monotone, so is the GPFF, and hence there must be a least fixpoint $\mathbf{p} = \mu(\Sigma_G)$ for any choice of signatures for G. The same is true for signatures that are specified syntactically.

Theorem 3.5. *Assume we are given a DFD G and, for each activator $n \in \mathcal{N}_G$, a syntactic signature $\mathbf{e}_n$. Then G can be extended to a consistently annotated DFD, with purpose labels*

$$n.purpose = [\![\mathbf{e}_n]\!] \qquad \text{for } n \in \mathcal{N}_G$$
$$f.purpose = \mu(\Sigma_G)(f) \qquad \text{for } f \in \mathcal{F}_G$$

where $\mu(\Sigma_G)$ is the least fixpoint of the GPFF Σ_G defined as

$$\Sigma_G(\mathbf{p})(f) = [\![\mathbf{e}_{s(f)}(f)]\!](\mathbf{p}|_{\text{in}(s(f))}).$$

This is the basis of the *purpose inference* algorithm (Alg. 2) described in §4.2.

Finally, note that every monotone function F on a finite bounded lattice $\mathcal{L}$ also has a *greatest fixpoint* $\nu(F)$, and that we might just as well compute a consistent flow annotation for a DFD G based on $\nu(\Sigma_G)$. The result is qualitatively different, though. A flow annotation computed via the least fixpoint of Σ_G is *conservative* in the sense that it starts from a minimal flow annotation $\bot$ – assuming data may not be used for any purpose – and progressively relaxes this restriction. A flow annotation computed via the greatest fixpoint of Σ_G, on the other hand, is *permissive* in the sense that it starts from a maximal flow annotation $\top$ – assuming data can be used for any purpose whatsoever – and progressively restricts this assumption.

Algorithm 1: Consistency Checking

input : A syntactically annotated DFD G
output: An error if G is inconsistent.

```
foreach f : m ⇝ n ∈ F_G do
    e ← e_n(f);
    ρ ← p_in(n);
    q ← [[e]]ρ;
    if p_f ⋢ q then
        Error: "The label on flow f is inconsistent with that
          computed by the signature e on node n."
```

4 ALGORITHMS AND IMPLEMENTATION

Here we present algorithms for checking the consistency of a given purpose-annotated DFD, and for inferring purpose labels on flows from syntactic signatures on nodes. Additionally, we describe our proof-of-concept implementation of these algorithms in our tool *PL-DFD*.

We focus on the core algorithms (Consistency Checking and Purpose Inference) implemented in *PL-DFD* and omit details about frontend operations such as parsing and pre-processing of DFDs, signatures, etc. since they are fairly standard. Furthermore, we pretend that the input to our algorithms (DFDs, purposes and syntactic signatures) are exactly as described in §3, even though the actual data structures are sightly more complicated. For example, we allow domain experts to define a collection of named (syntactic) flow signatures ahead of time, separately from DFDs, so that the same set of signatures can be reused by designers for different DFDs. Designers then annotate flows (rather than nodes) in DFDs with signature names (rather than lattice expressions) and must specify explicitly which input flows of the corresponding nodes are to be used as inputs to the signature in question. This representation of syntactically annotated DFDs is more flexible but also slightly more complex than the one given in §3.2.2. Since they are equivalent, we use the simpler representation when describing our algorithms.

4.1 Checking Consistency of Annotations

It is straightforward to check consistency of a syntactically annotated DFD G. As shown in the pseudo code for the *Consistency Checking* algorithm (Alg. 1), one simply compares the purpose annotation p_f of every flow f in G against the label computed from the (syntactic) signature of f and reports possible inconsistencies. In addition, our tool *PL-DFD* can suggest proper purpose labels to replace invalid ones based on the given privacy signatures.

4.2 Inferring Purpose Labels

The *Consistency Checking* algorithm works when DFD's flows are annotated with purpose labels. Instead of manually annotating a

Algorithm 2: Purpose Inference

input : A DFD G with syntactic signatures $\{\mathbf{e}_n\}_{n \in \mathcal{N}_G}$
output: A consistent syntactically annotated version of G

$\mathbf{q} \leftarrow \bot$;
repeat
  $\mathbf{p} \leftarrow \mathbf{q}$; $\mathbf{q} \leftarrow$ `PropagatePur`$(G, \mathbf{p})$
until $\mathbf{p} = \mathbf{q}$;
foreach $f \in \mathcal{F}_G$ **do** f.purpose $\leftarrow \mathbf{p}(f)$;

Function PropagatePur($G, \mathbf{p}$) – computes the GPFF of G.

input : A DFD G with syntactic signatures $\{\mathbf{e}_n\}_{n \in \mathcal{N}_G}$ and a flow annotation $\mathbf{p}$ for G
output: An updated flow annotation $\mathbf{q}$

$\mathbf{q} \leftarrow \{\}$; – *initialize* $\mathbf{q}$ *to the empty map*
foreach $f : m \rightsquigarrow n \in \mathcal{F}_G$ **do**
  $e \leftarrow \mathbf{e}_n(f)$;
  $\rho \leftarrow \mathbf{p}|_{\mathrm{in}(n)}$;
  $\mathbf{q}(f) \leftarrow [\![e]\!]_\rho$;
return $\mathbf{q}$;

DFD, designers may wish to automatically *infer* them, using the approached sketched in §3.2.3.

The *Purpose Inference* algorithm (Alg. 2) automates this procedure. It takes a DFD G annotated with privacy signatures (but not purpose labels) and computes a consistent flow annotation via a fixpoint procedure. Initially, flows are labeled with the bottom ($\bot$) purpose label – the most restrictive purpose ("don't use"). The initial flow annotation is then iteratively relaxed by applying the helper function PropagatePur, which propagates purpose labels from input flows to output flows (i.e. it implements the GPFF for G). The algorithm iteratively calls the function until a fixpoint is reached. This fixpoint procedure is guaranteed to terminate because the lattice $\mathcal{P}$ is finite, and Theorem 3.5 ensures that the resulting fixpoint is indeed a consistent purpose annotation for G.

Note that Alg. 2 is guaranteed to always infer a consistent purpose annotation for any DFD, but the result may look unexpected. In particular, there may be flows connected to external entities or data stores that have the purpose label $\bot = \{\}$, which is dubious since the label $\bot$ indicates that the data asset in question should not be used. An example of such a dubious flow is shown in Fig. 2b, where the "password" flow has label $\bot$ but is connected to the external entity Out_2. Such flows may indicate errors in a model and need to be checked carefully by designers to avoid privacy violations.

4.3 Proof-of concept tool

Our proposed framework for modeling purposes in DFDs comprises the two algorithms presented in the previous subsection. We implemented both algorithms in our *PL-DFD* tool. *PL-DFD* uses diagrams.net, a cross-platform, user-friendly, simple-to-use, open-source third-party application for drawing DFDs [15]. We utilize Henriksen's open source library [13] to provide additional support for manipulating DFDs. Since it is easy to import and export diagrams from/to XML format in diagrams.net, we represent DFD diagrams in an XML format. The checking result and the propagated purpose labels of DFD are displayed on CSV/Text file. Our tool is implemented in Python and has been experimented on a MacBook Pro. [3]

Table 1: Three privacy signature and their corresponding purpose propagation functions (PFFs).

Anonymize	Restrict	Forward
$x \mapsto \top$	$x, y \mapsto x \sqcap y$	$x \mapsto x$

5 EVALUATION

This section describes the evaluation of the implemented algorithms introduced in §4. We adopt and modify the DFD of a fictional Smart Speaker system, as analyzed by Lee et al. [17]. Then, we use our framework of (syntactically) annotated DFDs from §3 to extend the modified DFD with PFFs. For reproducibility, we have included all the DFD models (and results) that were used for evaluation in the source code repository.[3]

Smart Speaker. Smart speakers (such as Amazon Alexa and Google Echo) are widely spread voice recognition platforms that use AI software to translate users' voice commands into corresponding actions which are then autonomously carried out. For instance, playing music, adding reminders on users' cloud account, initiating a phone call via the user's connected smartphone, searching on the Internet, etc. Due to the increasing number of privacy concerns associated with similar devices calling for more robust data protection mechanisms [16], a fictitious smart speaker platform is an interesting case study to evaluate our privacy propagation framework.

Designer input. Figure 3 depicts a simplified diagram of our privacy enhanced Smart Speaker DFD. For a decomposed and detailed view of the inner-workings for each process of Figure 3 (including data stores), we refer the reader to the repository.[3]The regular DFD notation consists of external entities, processes, data flows, and data stores. The figure depicts what elements are involved when the device owner (external entity) downloads the app and installs it on their smartphone (process), logs in to the app, and starts using the speaker by interacting with it by sending voice commands (data flow). It also shows the data flows that are not directly visible to the device owner, i.e., when the speaker connects to the local WiFi and the provider forwarded data to third-party partners.

This diagram is free of privacy violations. In addition to the regular DFD notation elements, the designer is required to also: (i) annotate data flows with the purpose labels, (ii) compile a set of privacy signature definitions (see Table 1), (iii)annotate activators with signatures (e.g., the Speaker process *forwards* the password and WiFi ID to the Router), and (iv) define the permitted purpose(s) that the external entities are allowed to process the data for (annotations of external entities in Figure 3).

Table 1 lists the privacy signature definitions that a designer or domain expert needs to specify for *PL-DFD* (Algorithms 1 and 2) to work. For instance, in our case, the designer chose to model anonymization, restricting inputs (with possibly different purpose

[3]Source code available at https://github.com/alshareef-hanaa/PL-DFD.

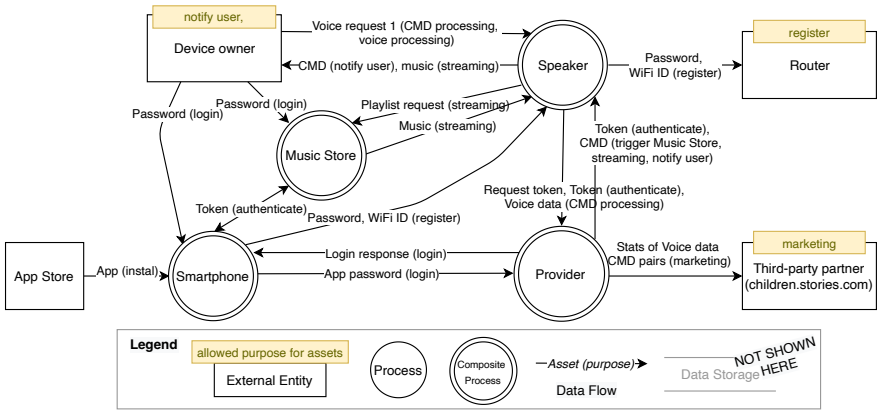


Figure 3: Context diagram of the smart speaker platform (without privacy violations)

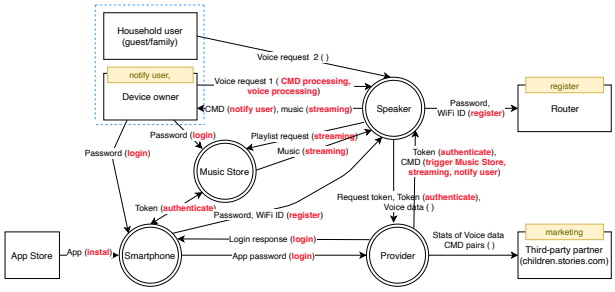
labels) and finally, forwarding data assets without performing any privacy computation. For brevity we omit the signature annotations (e.g., for one of the decomposed DFDs we included 45 instances of the defined privacy signatures) and refer the interested reader to the repository.

Scenario. The device owner can first download the mobile App for using the smart speaker. After the initial setup, the user will login to the App and connect the speaker to the local network (in Figure 3 the data flows with Password and WiFi ID are passed to the Router). The user can also login to the Music Store. The user activates the speakers' microphone and sends voice requests which are processes by the Provider, returning the corresponding command back to the speaker. The speaker then invokes the Music Store API provided capabilities to stream the desired content. Finally, the Provider may send certain aggregated statistics about their history of clienteles' requests to Third-party partners (e.g., children.stories.com).

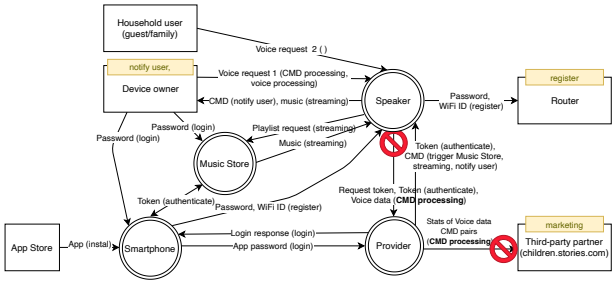
Method of evaluation. To illustrate the efficacy of the proposed algorithms we create another version of the DFD model for the smart speaker platform, which includes an additional external entity. We first run the Purpose Inference algorithm on a DFD model to infer purpose labels. Then, we run the Consistency Checking algorithm on an annotated DFD model to show two privacy violations. The privacy violations result from inconsistent purpose labels with propagation functions, leading to a privacy leak to an external entity.

Results. Figure 4 shows the effect of the two algorithms. Compared to the violation-free DFD presented in Figure 3, we add another external entity: a Household guest visiting the home of the device owner. The guest does not explicitly consent to any privacy policy, therefore "Voice request 2" has no allowed purpose.

Our inference algorithm (Alg. 2) is able to infer the purposes of all the DFD's flows, producing the DFD in (a). Let us assume the DFD designer does not consider the additional external entity ("Household user") and instead models the DFD as seen in Figure 4 (b). Alg. 1 can detect two privacy violations. The "Speaker" process has the ***restrict*** privacy signature attached to the incoming voice requests (1 and 2) into the output "Voice data". Upon propagation, the restrict signature is applied in the "Speaker" process for inputs "Voice request 1" and "Voice request 2", flowing into the output



(a) The DFD after inferring the purpose labels with Algorithm 2



(b) The annotated DFD after checking for violations with Algorithm 1

Figure 4: A graphical representation of the effect of Algorithms 1 and 2 on the models developed for the evaluation

"Voice data"[4]. The intersection of the purposes on the inputs is an empty set, which means it can not be used for any purpose. However, the purpose of the output flow is not an empty set ("CMD processing"). Therefore, the voice data should not be sent to the provider for further CMD computation, and our algorithm finds a violation. Within the same DFD, the "Provider" composite process ***forwards*** the voice data input to the voice data pairs on the output, which is then consumed by the Third-party. The Third-party is only authorized to consume data that is allowed to be used for marketing.

[4]Note that in the actual DSL, the data flows are *not* bundled together, every data asset is linked to a unique data flow.

However, upon propagation, the forward signature will copy the label on the output. Thus, the Third-party consumes data used for "CMD processing" (and not "marketing"), and our algorithm witnesses inconsistency. Such violations are detected by checking the consistency of the output flow purpose(s) and the permitted purpose(s) of an external entity (e.g. Third-party).

6 RELATED WORK

For a more comprehensive discussion on the challenges and open problems in privacy by design, we refer the reader to [19, 21] and the comprehensive ENISA report by Danezis et al. in [9].

As discussed in the introduction, DFDs have been already used in the context of PbD approaches [4, 5]. However, previous work has mostly focused on engineering approaches that do not provide correctness and completeness guarantees. For instance, Hoepman [14] has suggested that privacy can be addressed constructively during the creation of an architectural design by applying privacy *design strategies* Along the same lines, some catalogs of privacy design patterns have been proposed in the past, like for instance https://privacypatterns.org. Colesky et al. [8] suggest a set of privacy design tactics to be use as an intermediate refinement level between design strategies and design patterns. By addressing privacy from the very earl stages, these approaches allow software engineers to avoid potential privacy issues early on. The design models deriving from the adoption of the above-mentioned approaches can undergo analysis by using, for instance, threat modeling approaches like the one proposed by Sion et al. [20] These approaches are complementary to our work, which takes a more formal stance to the problem of specifying and analyzing privacy in design models. The adoption of formal methods in PbD has been advocated in the past by Antignac and Le Métayer [3].

Basin et al. [6] have proposed an approach to check GDPR compliance in *business process models*, which are somehow related to flow diagrams (even though the semantics of processes is quite different in the two types of diagrams). In particular, they assign "purpose" to a "process", automatically generate privacy policies and can detect violations of the principle of data minimization. Their work is complementary to ours (which focuses on purpose limitation rather than data minimization). However, their paper also highlights the difficulty of representing the notion of *purpose* at the level of software entities and problematizes the possibility of performing fully automated checks for GDPR compliance. Although we agree with this position, in this paper we try to push the envelope of automatization for a specific privacy property.

Finally, our work is inspired by the work of Tuma et al. [22], which propose an approach to analyze information flow policies in DFD models. Their work focuses on data confidentiality and, partly, on data integrity. Similarly to our work, they define four types of contracts for process nodes (in the shape of labels) and introduce propagation rules for the data assets on the flows traversing the annotated processes. The work of Tuma et al. focuses on detecting information leakage (security), while this paper focuses on purpose limitation (privacy).

7 CONCLUSIONS

By leveraging the leverage the DFD notation, this paper has proposed a formal approach to the specification of purpose limitation in design models. In particular, we have presented a framework to annotate process nodes with semantically defined purpose labels and have introduced a collection of algorithms to (i) compute the propagation of such labels in the model graph and (ii) check the consistency of the information flows with respect to said labels. The contributions in this work complement the work on DFD by Antignac et al. [5] and Alshareef et al. [2].

In the future, we would like to investigate the usability of our framework by software developers and designers. For that, it would be useful to improve our implementation so that rich explanations could be provided to the developers in case of violations, possibly with suggestions on how to fix the design flaws.

ACKNOWLEDGMENTS

This work was partly funded by the Saudi Cultural Office in Berlin, the EU Horizon 2020 program via under Grant 952647 (AssureMOSS), and the Swedish Research Council under Grant 2018-04230.

REFERENCES

[1] Hanaa Alshareef, Sandro Stucki, and Gerardo Schneider. 2021. Refining Privacy-Aware Data Flow Diagrams. In *SEFM'21*. Springer, 121–140.
[2] Hanaa Alshareef, Sandro Stucki, and Gerardo Schneider. 2021. Transforming Data Flow Diagrams for Privacy Compliance. In *MODELSWARD'21*. 207–215.
[3] Thibaud Antignac and Daniel Le Métayer. 2014. Privacy by Design: From Technologies to Architectures. In *APF'14*. Springer, 1–17.
[4] Thibaud Antignac, Riccardo Scandariato, and Gerardo Schneider. 2016. A Privacy-Aware Conceptual Model for Handling Personal Data. In *ISoLA'16*. 942–957.
[5] Thibaud Antignac, Riccardo Scandariato, and Gerardo Schneider. 2018. Privacy Compliance via Model Transformations. In *IWPE'18*. IEEE, 120–126.
[6] David Basin, Søren Debois, and Thomas Hildebrandt. 2018. On purpose and by necessity: compliance under the GDPR. In *FC'18*. Springer, 20–37.
[7] Ann Cavoukian. 2012. Privacy by design: origins, meaning, and prospects for assuring privacy and trust in the information era. In *Privacy Protection Measures and Tech. in Business Org.* IGI Global, 170–208.
[8] Michael Colesky, Jaap-Henk Hoepman, and Christiaan Hillen. 2016. A Critical Analysis of Privacy Design Strategies. In *Sec. and Priv Workshop*. IEEE, 33–40.
[9] George Danezis, Josep Domingo-Ferrer, Marit Hansen, Jaap-Henk Hoepman, Daniel Le Métayer, Rodica Tirtea, and Stefan Schiffner. 2015. Privacy and Data Protection by Design. ENISA Report.
[10] Alan Dennis, Barbara Haley Wixom, and Roberta M Roth. 2018. *Systems analysis and design*. John wiley & sons.
[11] European Commission. 2016. *General Data Protection Regulation (GDPR)*. Regulation 2016/679. European Commission.
[12] E.D Falkenberg, R Van Der Pols, and Th.P Van Der Weide. 1991. Understanding process structure diagrams. *Information Systems* 16, 4 (1991), 417 – 428.
[13] Michael Henriksen. 2018. Draw.io libraries for threat modeling diagrams. https://github.com/michenriksen/drawio-threatmodeling
[14] Jaap-Henk Hoepman. 2014. Privacy design strategies. In *IFIP International Information Security Conference*. Springer, 446–459.
[15] JGraph Ltd. 2022. diagrams.net. https://www.diagrams.net/.
[16] Jacob Leon Kröger, Otto Hans-Martin Lutz, and Philip Raschke. 2019. Privacy implications of voice and speech analysis–information disclosure by inference. In *IFIP Int. Summer School*. Springer, 242–258.
[17] Jiseop Lee, Sooyoung Kang, and Seungjoo Kim. 2019. Study on the smart speaker security evaluations and countermeasures. In *Advanced Multimedia and Ubiquitous Engineering*. Springer, 50–70.
[18] Marie Caroline Oetzel and Sarah Spiekermann. 2014. A systematic methodology for privacy impact assessments: a design science approach. *European Journal of Information Systems* 23, 2 (2014), 126–150.
[19] Gerardo Schneider. 2018. Is Privacy by Construction Possible?. In *ISoLA'18*. Springer, 471–485.
[20] Laurens Sion, Dimitri Van Landuyt, Kim Wuyts, and Wouter Joosen. 2019. Privacy risk assessment for data subject-aware threat modeling. In *IEEE Security and Privacy Workshops*.
[21] Pagona Tsormpatzoudi, Bettina Berendt, and Fanny Coudert. 2015. Privacy by Design: From Research and Policy to Practice - the Challenge of Multi-disciplinarity. In *APF'15*. Springer, 199–212.
[22] Katja Tuma, Riccardo Scandariato, and Musard Balliu. 2019. Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis. In *ICSA'19*. IEEE, 191–200.
[23] Kim Wuyts, Riccardo Scandariato, and Wouter Joosen. 2014. Empirical evaluation of a privacy-focused threat modeling methodology. *J. of Syst. and Soft.* (2014), 122–138.

A PROOF OF LEMMA 3.4

Lemma 3.4 (monotonicity). *The function $[\![e]\!]\colon \mathcal{P}^V \to \mathcal{P}$ is monotone for any lattice expression e, i.e. if $\rho \subseteq \sigma$, then $[\![e]\!]_\rho \subseteq [\![e]\!]_\sigma$.*

Proof. The proof relies on monotonicity of $\cup$ and $\cap$. We know from basic lattice theory that the least upper bound (LUB) $\sqcup$ and greatest lower bound (GLB) $\sqcap$ operations are monotone in any (semi-)lattice. Monotonicity of $\cup$ and $\cap$ follows as a special case for powerset lattices. Let us quickly recap the general proof for $\sqcap$; the one for $\sqcup$ is dual. Assume $x \sqsubseteq x'$ and $y \sqsubseteq y'$. Because $x \sqcap y$ is a lower bound of x and y, we have $x \sqcap y \sqsubseteq x \sqsubseteq x'$ and $x \sqcap y \sqsubseteq y \sqsubseteq y'$, so $x \sqcap y$ is also a lower bound of x' and y'. Since $x' \sqcap y'$ is the greatest lower bound of x' and y', we must have $x \sqcap y \sqsubseteq x' \sqcap y'$. Hence $\sqcap$ is monotone in both its arguments.

The proof of Lemma 3.4 is now an easy induction on the structure of lattice expressions e. We show the cases for variables x, constants p and GLBs $e_1 \sqcap e_2$, the others are similar.

- **Case $e = x$.** Assume $\rho \subseteq \sigma$, i.e. $\rho(x) \subseteq \sigma(x)$ for all $x \in V$. Then $[\![x]\!]_\rho = \rho(x) \subseteq \sigma(x) = [\![x]\!]_\sigma$.
- **Case $e = p$.** Trivial: $[\![p]\!]_\rho = p = [\![p]\!]_\sigma$ for any ρ and σ.
- **Case $e = e_1 \sqcap e_2$.** By the induction hypothesis, $[\![e_1]\!]$ and $[\![e_2]\!]$ are monotone, i.e. if $\rho \subseteq \sigma$, then $[\![e_1]\!]_\rho \subseteq [\![e_1]\!]_\sigma$ and $[\![e_2]\!]_\rho \subseteq [\![e_2]\!]_\sigma$. By monotonicity of GLBs,

$$\begin{aligned} [\![e_1 \sqcap e_2]\!]_\rho &= [\![e_1]\!]_\rho \cap [\![e_2]\!]_\rho \\ &\subseteq [\![e_1]\!]_\sigma \cap [\![e_2]\!]_\sigma && \text{(by monotonicity of } \cap) \\ &= [\![e_1 \sqcap e_2]\!]_\sigma . \end{aligned}$$

□