



Transforming Data Flow Diagrams for Privacy Compliance

Hanaa Alshareef¹, Sandro Stucki²^a and Gerardo Schneider²^b

¹Chalmers University of Technology, Gothenburg, Sweden

²University of Gothenburg, Gothenburg, Sweden

Keywords: Privacy by Design, Data Flow Diagrams, GDPR.

Abstract: Most software design tools, as for instance Data Flow Diagrams (DFDs), are focused on functional aspects and cannot thus model non-functional aspects like privacy. In this paper, we provide an explicit algorithm and a proof-of-concept implementation to transform DFDs into so-called Privacy-Aware Data Flow Diagrams (PA-DFDs). Our tool systematically inserts privacy checks to a DFD, generating a PA-DFD. We apply our approach to two realistic applications from the construction and online retail sectors.

1 INTRODUCTION

The *European General Data Protection Regulation* (GDPR) imposes stringent constraints on how individuals' personal data are to be collected and processed, stipulating heavy penalties in case of violations (European Commission, 2016). Complying the regulation is a hard task and software engineers trying to meet the required data protection principles often face a conflict between system and privacy requirements (Oetzel and Spiekermann, 2014).

An additional difficulty is that privacy does not refer to one particular property but rather to a set of properties, including confidentiality, secrecy, data minimisation (DM), privacy impact assessment (PIA), user consent, the right to be forgotten, purpose limitation, etc. So, it does not make sense to talk about "privacy compliance" but rather to refer to specific privacy properties. But even when restricted to a specific privacy property, verifying privacy compliance is in general undecidable (Tsormpatzoudi et al., 2015; Schneider, 2018).


We therefore advocate the *Privacy by Design* (PbD) principle (Cavoukian, 2012), in which any (computerised) personal data processing environment should be designed taking privacy into account from the very beginning of the (software) development process. It has been argued that PbD is more tractable than retrofitting legacy software for privacy compliance (see e.g. Danezis et al., 2015).


Still, the implementation of privacy principles such as PbD, PIA or DM requires a lot of work from software engineers and developers: they consider such principles to be overly complicated and impractical, and they lack the necessary knowledge to implement them (Senarath and Arachchilage, 2018; Sirur et al., 2018; Freitas and Mira da Silva, 2018). Hence, despite having been advocated since the mid-1990s, PbD has gained momentum only in recent years, mostly due to the GDPR.

An example is the work by Antignac et al. (2016, 2018), who propose an approach to automatically add privacy checks already at the design level. The idea is based on model transformations, enhancing *Data Flow Diagrams* (DFDs) with checks for specific privacy concepts, notably concerning retention time and purpose limitation for each operation on sensitive (personal) data (storage, forwarding, and processing of data). 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" but provide neither a full algorithm nor a reference implementation. The main purpose of our paper is to provide these missing pieces.

In summary, we make the following contributions.

1. We give algorithms to check and automatically transform DFDs into PA-DFDs. We identified

^a <https://orcid.org/0000-0001-5608-8273>

^b <https://orcid.org/0000-0003-0629-6853>

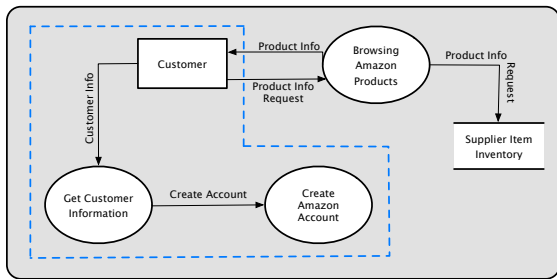


Figure 1: Example of a DFD: high-level design of part of the e-store ordering system.

some ambiguities and inaccuracies in the original description given in the hotspots' translation by Antignac et al. (2016, 2018). (Section 3).

2. We provide an open-source Python implementation of our algorithms,¹ which processes DFD diagrams in an XML format compatible with the popular `draw.io` platform (Section 3).
3. We evaluate our algorithms on two case studies: an automated payment system and an online retail system (Section 4).

2 PRELIMINARIES

We recall here relevant GDPR concepts, the definition of DFDs, as well as the transformation into PA-DFDs given by Antignac et al. (2018).

GDPR. The European *General Data Protection Regulation* (GDPR) contains 99 articles regulating *personal data* processing. The GDPR is organised around a number of key concepts, most notably its seven *principles* of personal data processing, the *rights* of data subjects and six *lawful grounds* for data processing operations. Relevant to this paper are the principles of *purpose limitation* (data may only be used for purposes to which the data subject consented) and *accountability*, as well as the *right to be forgotten* and the lawful ground of *consent*. See (European Commission, 2016) and Hert and Papakonstantinou (2016) for more details on the GDPR.

Data Flow Diagrams (DFDs). A *data flow diagram* (DFD) is a graphical representation of how data flows among software components. As shown in Fig. 1, DFDs are composed of *activators* and *flows*. Activators can be *external entities* (rectangles), *processes* (ellipses) and *data stores* (double horizontal lines). Processes may represent detailed low-level operations or complex high-level functionality that

¹<https://github.com/alshareef-hanaa/PA-DFD-Paper>

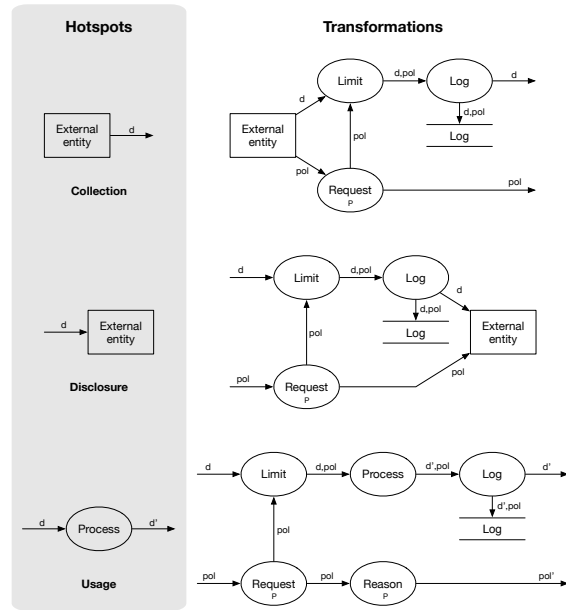


Figure 2: Selection of B-DFD hotspots and corresponding PA-DFD elements (Antignac et al., 2018).

could be refined into sub-processes (the latter are drawn as double-lined ellipses). Data *flow* is represented by arrows. We chose DFDs since they are a widely used for modelling digital systems and for security and privacy analysis (Shostack, 2014; Wuyts et al., 2014).

Antignac et al. (2016, 2018) extended DFDs with a *data deletion* type of flow and a data structure to specify personal data: (i) the *owner* of personal data, (ii) the *purpose* for which the data can be used consented by the data subject, and (iii) the *retention* time for the data. This extension is referred to as *Business-oriented DFD* (B-DFD).

Adding Privacy Checks to DFDs. Antignac et al. (2016, 2018) aimed at (automatically) add privacy checks to a B-DFD, obtaining a *Privacy-Aware Data Flow Diagram* (PA-DFD) which contains relevant privacy checks for purpose limitation and retention time, as well as to ensure accountability and policy management. They defined *hotspots* in the B-DFD to perform the transformation compositionally.

The left-hand side of Fig. 2 shows three types of hotspots, each defined by a pattern of activators and flows that corresponds to a basic data processing operation, such as “collection”, “disclosure”, etc. The right-hand side of Fig. 2 shows, for each B-DFD hotspot, the corresponding PA-DFD containing new activators and flows for specific privacy mechanisms.

Tables 1 and 2 in Antignac et al. (2018) describes the privacy properties of interest for each hotspot,

(*proc*) and data stores (*db*); flows are classified as either plain data flows (*pf*) or data deletions (*df*). Fig. 1 shows an example of a B-DFD with five activators (an external entity, a datastore and three processes) that are connected by plain flows.

Definition 2. We define the set of *data node types* as $\mathcal{T}_{dn} = \{ext, proc, db\}$ and the set of *raw flow types* as $\mathcal{T}_{rf} = \{pf, df\}$. A (*raw*) *B-DFD* is an attributed multigraph G with activators as nodes and flows as edges, and where we fix \mathcal{A} and \mathcal{V} to be $\mathcal{A} = \{type\}$, $\mathcal{V} = \mathcal{T}_{dn} \uplus \mathcal{T}_{rf}$. In addition, every activator and flow must have a type, i.e., $n.type \in \mathcal{T}_{dn}$ and $f.type \in \mathcal{T}_{rf}$ must be defined for all n and f .

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

Well-formed B-DFDs differ from raw B-DFDs primarily in the choice of flow types. Flows are typed based on their source and target activators. Only some combinations of sources, targets and flow types are valid. They are shown on the left-hand side of Fig. 5. If a flow does not conform to one of these six cases, it is *ill-typed* and will be rejected by our type inference algorithm. In addition to these flow typing constraints, we adopt some common rules from the DFD literature for well-formed B-DFDs: diagrams may 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. Falkenberg et al., 1991; Dennis et al., 2018).

Definition 3. We define the set of *data flow types* as $\mathcal{T}_{df} = \{in, out, comp, store, read, delete\}$. A *well-formed B-DFD* is an attributed multigraph G , where $\mathcal{A} = \{type\}$ and $\mathcal{V} = \mathcal{T}_{dn} \uplus \mathcal{T}_{df}$. In addition, flows and activators are subject to the following conditions:

- $n.type \in \mathcal{T}_{dn}$ and $f.type \in \mathcal{T}_{df}$;
- if $f: n \rightsquigarrow_{in} m$ then $n: ext$ and $m: proc$;
- if $f: n \rightsquigarrow_{out} m$ then $n: proc$ and $m: ext$;
- if $f: n \rightsquigarrow_{comp} m$ then $n: proc, m: proc$ and $n \neq m$;
- if $f: n \rightsquigarrow_{store} m$ then $n: proc$ and $m: db$;
- if $f: n \rightsquigarrow_{read} m$ then $n: db$ and $m: proc$;
- if $f: n \rightsquigarrow_{delete} m$ then $n: proc$ and $m: db$;
- if $n: proc$ then $n \in S(G)$ and $n \in T(G)$
- if $n: ext$ or $n: db$ then $n \in S(G)$ or $n \in T(G)$

The Type-inference algorithm (Algorithm 1) detects and reports any ill-formed flows (lines 1–12) and activators (lines 13–19). If type inference is successful, the resulting well-formed B-DFD can safely be transformed into a PA-DFD.

Algorithm 1: Type-inference.

```

input  : A raw B-DFD  $G$ 
output : A well-formed version of  $G$ 
1 foreach  $f: m \rightsquigarrow n \in \mathcal{F}$  do
2   if  $f.type = pf$  then
3     if  $m: ext \wedge n: proc$  then  $f.type \leftarrow in$ ;
4     else if  $m: proc \wedge n: ext$  then  $f.type \leftarrow out$ ;
5     else if  $m: proc \wedge n: proc \wedge m \neq n$  then
6        $f.type \leftarrow comp$ 
7     else if  $m: proc \wedge n: db$  then  $f.type \leftarrow store$ ;
8     else if  $m: db \wedge n: proc$  then  $f.type \leftarrow read$ ;
9     else  $f$  is ill-formed;
10  else if  $f.type = df$  then
11    if  $m: proc \wedge n: db$  then  $f.type \leftarrow delete$ ;
12    else  $f$  is ill-formed;
13 foreach  $n \in \mathcal{N}$  do
14   if  $n: proc \wedge (n \notin S(G) \vee n \notin T(G))$  then
15      $n$  is ill-formed
16   else if  $n: ext \wedge (n \notin S(G) \wedge n \notin T(G))$  then
17      $n$  is ill-formed
18   else if  $n: db \wedge (n \notin S(G) \wedge n \notin T(G))$  then
19      $n$  is ill-formed
    
```

3.2 Transformation

Well-formed B-DFDs are guaranteed to be well-formed, but they do not yet contain any explicit privacy checks. They are introduced by Algorithm 2, which transforms each flow in the well-formed B-DFD into a set of corresponding PA-DFD elements (see Fig. 5). These PA-DFD elements represent the functionality for enforcing purpose limitation, retention time, accountability and policy management.

First we add *reason* activators for each process in the well-formed B-DFD. These activators are linked to each other by a special *partner* attribute. Each *reason* activator is assigned to exactly one process via this attribute. Likewise, we add a new *policy_db* activator to each data store in the well-formed B-DFD and link them via their *partner* attributes. The second phase of the algorithm transforms each flow based on its type (i.e., the hotspot that it belongs to). We use dedicated helper procedures to transform each flow type (e.g., *addInElems*, which transforms *in* flows). The auxiliary procedures introduce the necessary activators and flows for checking and logging each data flow. The *partner* attributes of the original flow's source and target are used to identify the activators that supply and transfer the required policy values.

As with B-DFDs, we use attributed graphs to represent PA-DFDs formally.

Definition 4. Define the set of *policy node types* as $\mathcal{T}_{pn} = \{limit, request, reason, policy_db\}$ and the set

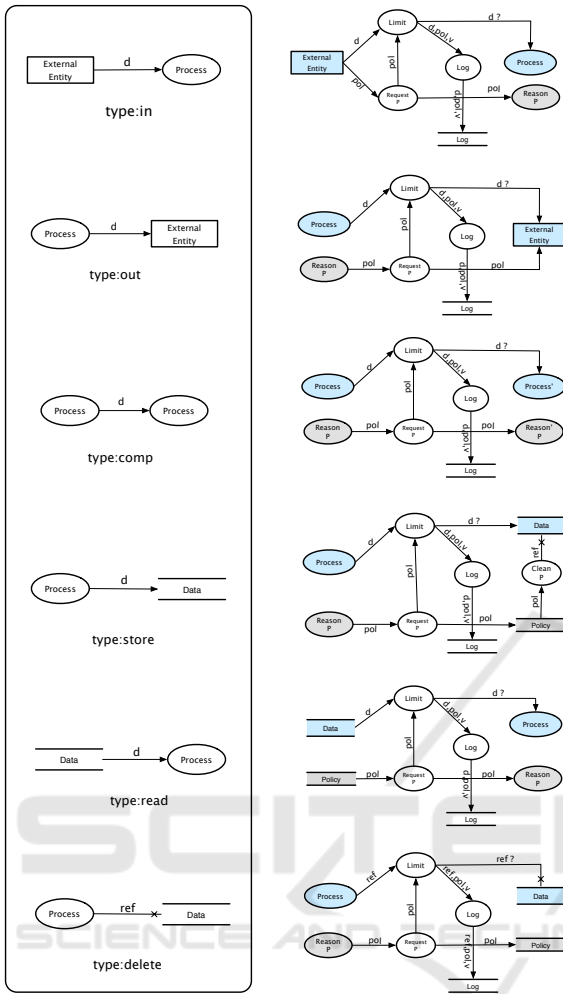


Figure 5: Well-formed B-DFD and the updated corresponding PA-DFD elements.

of *admin node types* as $\mathcal{T}_{an} = \{\log, \log_db, \text{clean}\}$. A *PA-DFD* is an attributed graph G , where $\mathcal{A} = \{\text{type}, \text{partner}\}$ and $\mathcal{V} = \mathcal{T}_{dn} \uplus \mathcal{T}_{pn} \uplus \mathcal{T}_{an} \uplus \mathcal{T}_{rf} \uplus \mathcal{N}$. In addition, the following must hold:

- $n.\text{type} \in \mathcal{T}_{dn} \uplus \mathcal{T}_{pn} \uplus \mathcal{T}_{an}$ and $f.\text{type} \in \mathcal{T}_{rf}$;
- if $n.\text{partner}$ is defined, then $n.\text{partner} \in \mathcal{N}$

In principle, the flows of PA-DFDs ought to be subject to similar typing conditions as those for well-formed B-DFDs. Following the principle used for well-formed B-DFDs, we could type each flow based on the types of its source and target. For example, the flows connecting *request* to *limit* activators could be given type *reqlim*. This would result in eighteen new flow types. To simplify presentation, we instead use just two flow types for PA-DFDs as we did for raw B-DFDs: *plain flows* (*pf*) and *deletion flows* (*df*).

Algorithm 2: Transformation.

input : A well-formed B-DFD G
output : A PA-DFD

- 1 **foreach** $n \in \mathcal{N}$ **do**
- 2 **if** n : *proc* **then**
- 3 add a new node m : *reason* to G ;
- 4 $m.\text{partner} \leftarrow n$; $n.\text{partner} \leftarrow m$
- 5 **if** n : *db* **then**
- 6 add a new node m : *policy_db* to G ;
- 7 $m.\text{partner} \leftarrow n$; $n.\text{partner} \leftarrow m$
- 8 **foreach** $f \in \mathcal{F}$ **do**
- 9 **if** f : *in* **then** *addInElems* (n, f, G);
- 10 **if** f : *out* **then** *addOutElems* (n, f, G);
- 11 **if** f : *comp* **then** *addCompElems* (n, f, G);
- 12 **if** f : *store* **then** *addStoreElems* (n, f, G);
- 13 **if** f : *read* **then** *addReadElems* (n, f, G);
- 14 **if** f : *delete* **then** *addDeleteElems* (n, f, G);

Comparison of Transformation Rules. The transformation rules presented in Fig. 2 have a few subtle but important shortcomings that are addressed in our Transformation algorithm.

First, the rules do not explain how activators with multiple input and output flows are to be transformed. Note that all activators on the left of Fig. 2 have at most one incoming or outgoing flow. It is unclear which of the new activators and flows shown on the right should be added only once per rule application, and which need to be instantiated for every incoming or outgoing flow. We solve this problem by splitting the transformation into two distinct steps. In a first step, we create *reason* and *policy_db* nodes as *partners* for processes and data stores. In the second step, each original flow is equipped with the activators and flows implementing the new privacy checks. This two-step approach cleanly separates the per-activator and per-flow aspects of each rule.

Second, the *limit* and *log* activators in the original rules are set up in a problematic way. Every *limit* activator is followed by a *log* activator that receives both a policy and a data value. The *log* activator logs both values and forwards the data value to downstream activators. But what if a privacy violation occurs? The *limit* activator should inhibit such violations by blocking unintended flows, passing on only policy-compliant data values. Hence, policy violation events never reach the *log* activator, and are therefore not logged. This seems highly problematic. Alternatively *limit* nodes could pass on *all* data and policy values (irrespective of violations), leaving the *log* activator to perform the actual filtering. But why have separate *limit* and *log* activators in the first place then? We resolve this ambiguity by connecting *limit* activators directly to the downstream activators and,

Schaefer et al. (2018) present a definition of rules for achieving *Confidentiality-by-Construction*, where functional specifications are replaced by confidentiality specifications listing which variables contain secrets. Though the approach seems interesting, it has (to the best of our knowledge) not been implemented.

Tuma et al. (2019) analyse information flow policies at the modelling level. They focus on data confidentiality and integrity, and introduce a graphical notation based on DFDs to algorithmically detect design flaws “in the form of violations of the intended security properties”. They provide an Eclipse-based implementation. Their approach is also based on DFDs but has different objectives: while we focus on the implementation of model transformation for specific privacy checks, Tuma et al. focus on the detection of design flaws associated with security properties.

Our paper distinguishes itself in that none of the above has taken the approach to automatically add privacy checks to design models.

6 CONCLUSIONS

We have provided algorithms to automatically translate DFD models into privacy-aware DFDs (PA-DFDs) as well as a proof-of-concept implementation integrated into a graphical tool for drawing DFDs. Obtaining the algorithms (from the existing conceptual transformation) was not easy as some aspects of the transformation were subtle and ambiguous not allowing for a direct implementation. We have addressed these conceptual flaws and evaluated them through two case studies: an automated payment system and an online retail system.

One limitation of our approach is that the diagrams resulting from our transformation can be large, making it difficult to visualise them. That said, the intended use of this tool is as an intermediate step in the design and development process, so the software architect can still be able to inspect (and modify) only small and relevant subsets of the PA-DFD. Our next step is to implement an algorithm to automatically synthesise a template from the PA-DFD in Java or Python. We will provide the programmer with predefined libraries to be used as building blocks for implementing such privacy checks.

REFERENCES

- Alshareef, H., Stucki, S., and Schneider, G. (2020). Transforming data flow diagrams for privacy compliance (long version). *CoRR*, abs/2011.12028.
- Antignac, T., Scandariato, R., and Schneider, G. (2016). A privacy-aware conceptual model for handling personal data. In *ISoLA'16*, pages 942–957.
- Antignac, T., Scandariato, R., and Schneider, G. (2018). Privacy compliance via model transformations. In *IWPE'18*, pages 120–126. IEEE.
- Basin, D., Debois, S., and Hildebrandt, T. (2018). On purpose and by necessity: compliance under the GDPR. In *FC'18*, pages 20–37. Springer.
- Cavoukian, A. (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.*, pages 170–208. IGI Global.
- Chong, H.-Y. and Diamantopoulos, A. (2020). Integrating advanced technologies to uphold security of payment: Data flow diagram. *Automation in Construction*, 114:103–158.
- Danezis, G., Domingo-Ferrer, J., Hansen, M., Hoepman, J.-H., Le Métayer, D., Tirtza, R., and Schiffner, S. (2015). Privacy and data protection by design. ENISA Report.
- Dennis, A., Wixom, B. H., and Roth, R. M. (2018). *Systems analysis and design*. John Wiley & sons.
- draw.io (2019). draw.io. <https://www.draw.io/>.
- European Commission (2016). General data protection regulation (GDPR). Regulation 2016/679, European Commission.
- Falkenberg, E., Pols, R. V. D., and Weide, T. V. D. (1991). Understanding process structure diagrams. *Information Systems*, 16(4):417 – 428.
- Freitas, M. and Mira da Silva, M. (2018). GDPR compliance in SMEs: There is much to be done. *J. Inform. Systems Eng.*, 3(4):30.
- Henriksen, M. (2018). Draw.io libraries for threat modeling diagrams. <https://github.com/michenriksen/drawio-threatmodeling>.
- Hert, P. D. and Papakonstantinou, V. (2016). The new general data protection regulation: Still a sound system for the protection of individuals? *Computer Law & Security Review*, 32(2):179–194.
- Oetzel, M. C. and Spiekermann, S. (2014). A systematic methodology for privacy impact assessments: a design science approach. *European Journal of Information Systems*, 23(2):126–150.
- Schaefer, I., Runge, T., Knüppel, A., Cleophas, L., Kourie, D., and Watson, B. W. (2018). Towards confidentiality-by-construction. In *ISoLA'18*. Springer.
- Schneider, G. (2018). Is privacy by construction possible? In *ISoLA'18*, pages 471–485. Springer.
- Senarath, A. and Arachchilage, N. A. (2018). Why developers cannot embed privacy into software systems? an empirical investigation. In *EASE'18*, pages 211–216.
- Shostack, A. (2014). *Threat modeling: Designing for security*. John Wiley & Sons.
- Sirur, S., Nurse, J. R., and Webb, H. (2018). Are we there yet? Understanding the challenges faced in complying with the general data protection regulation (GDPR). In *MPS'18*, pages 88–95. ACM.
- Tsormpatzoudi, P., Berendt, B., and Coudert, F. (2015). Privacy by design: From research and policy to prac-

- tice - the challenge of multi-disciplinarity. In *APF'15*, pages 199–212. Springer.
- Tuma, K., Scandariato, R., and Balliu, M. (2019). Flaws in flows: Unveiling design flaws via information flow analysis. In *ICSA'19*, pages 191–200. IEEE.
- Wuyts, K., Scandariato, R., and Joosen, W. (2014). Empirical evaluation of a privacy-focused threat modeling methodology. *J. of Syst. and Soft.*, 96:122–138.

