Graph Theory for Consent Management: A New Approach for Complex Data Flows

Dorota Filipczuk*
Microsoft
Dronning Eufemias gate 71
0194 Oslo, Norway
dorotaf@acm.org

Enrico H. Gerding
University of Southampton
University Road
Southampton SO17 1BJ, UK
eg@ecs.soton.ac.uk

George Konstantinidis
University of Southampton
University Road
Southampton SO17 1BJ, UK
G.Konstantinidis@soton.ac.uk

ABSTRACT

Through legislation and technical advances users gain more control over how their data is processed, and they expect online services to respect their privacy choices and preferences. However, data may be processed for many different purposes by several layers of algorithms that create complex data workflows. To date, there is no existing approach to automatically satisfy fine-grained privacy constraints of a user in a way which optimises the service provider's gains from processing. In this article, we propose a solution to this problem by modelling a data flow as a graph. User constraints and processing purposes are pairs of vertices which need to be disconnected in this graph. We show that, in general, this problem is NP-hard and we propose several heuristics and algorithms. We discuss the optimality versus efficiency of our algorithms and evaluate them using synthetically generated data. On the practical side, our algorithms can provide nearly optimal solutions for tens of constraints and graphs of thousands of nodes, in a few seconds.

1. INTRODUCTION

Personal data processing is at the core of many computing systems. In some complex ones, such as those owned by Netflix, Meta or Amazon, users' data is automatically processed by microservices. Typically, personal data enters the system through front-end applications, which send requests to a subset of services. There, each individual service performs a specific business function independent from the rest of the services, often producing predictions or inferences that are in turn consumed by other services. For instance, in a social media system, the user's location may be used by a service responsible for processing user profile information that then sends this data to a usage analytics service and to a service for suggesting groups to join. Consequently, the inferences made by the usage analytics service may be used by an ads-ranking service and the predicted groups of user's interest may be used by a service recommending products to buy. These services next call other services, creating data flows that span over multiple layers of computation. The ultimate goal of this data processing is to eventually satisfy certain business goals by which the service provider gains utility e.g., the ads ranking service is used to increase revenue through personalised advertisement and the product recommendation service may serve the purpose of collecting commission on product sale.

However, individuals should have control over how their data is used. Particularly, in certain regulatory frameworks such as the GDPR [12], unless there exists another legal basis, user's consent is necessary to legally allow personal data to be processed. By opting out of data processing, e.g., not allowing their location data to be used for personal advertisement or product recommendation, users put constraints on the data flow. Enforcing these constraints affects the utility of the service provider who would like the utility loss minimised. What complicates the task even more is its large scale: in modern computing systems, there may be many stages of data processing, in a data flow that involves hundreds of nodes. In fact, Meta's microservice topology contains over 12 million service instances and over 180,000 communication edges between services [15]. In such cases, stopping the data flow to the ads-ranking service or the product recommendation service may affect the quality of the inferences that other services use, and thus, the utility of the service provider.

In this article, we propose a novel approach to finding optimal ways of satisfying privacy constraints automatically while minimising the utility loss for the service provider. We model the data flow as a graph and privacy constraints as pairs of vertices in the graph. Our problem definition is generic, allowing the utility of a purpose node to be a black-box function of the subgraph connected to that node. We formulate the problem as an optimisation problem, where pairs of graph vertices must be disconnected such that utility is maximised (Section 2) and demonstrate it on an example use case (Section 3). We then present a natural instantiation of the generic problem where the utility functions are linearly additive (Section 4).

We present five generic heuristics for implementing the privacy constraints into data workflows (Section 5) with an optimal or approximately optimal global utility, depending on the actual utility function used. We further analyse the (non-)optimality of our heuristics in Section 6. For our experiments (Section 7) we implement the aforementioned linearly additive instance of the problem. Notably, we show that, although computing the optimal solution can be very time-consuming, the proposed heuristics can provide good and efficient approximation alternatives. Our ap-

 DOI: http://dx.doi.org/10.48786/edbt.2023.61.

*This research was conducted while the author was at the University of Southampton.

proach opens a range of new problems that we discuss in Section 8. Finally we juxtapose our approach with related work in Section 9.

2. PROBLEM FORMULATION

We focus on what we call the *Consented Data Workflow* problem (CDW): given user constraints expressed in terms of the vertices that they do not wish to be connected, find a subgraph of the original workflow where these constraints are satisfied. In the face of alternative solutions, the optimal one should minimise the utility loss for the service provider from applying the constraints. In other words, we are looking for the utility-maximising solution subject to the users' privacy constraints.

Formally, our data processing model is a directed graph $G = (V, E)$ with a set of edges E representing the data flow and a set of vertices V representing the stages of data processing. We distinguish three kinds of vertices, i.e. $V = V^U \cup V^A \cup V^P$, where V^U is a set of user data vertices, which represent the types of data collected directly from the user, V^A is a set of algorithm vertices, which represent data processing algorithms that take one or more data types as input, and V^P is a set of purpose vertices, which represent the end goals of data processing.

Furthermore, satisfying the given purposes is what brings service providers utility. For any vertex in V^P, we will have an associated utility that reflects the value processing that this purpose brings to service providers. In practice, the service providers' valuation depends on factors such as the accuracy of the datasets used as the input to the processing algorithms [20, 14]. In particular, where data processing is a multi-stage process, the utility is affected by all stages and all datasets processed for the purpose.

Therefore, in order to calculate the utility of data processing for a given purpose, in our model we look for all vertices and edges that carry the data workflow to the given purpose vertex. Formally, we say that a vertex $v_i \in V$ is *reachable* from a vertex $v_j \in V$ if there exists a path in G, $\{(v_1, v_2),(v_2, v_3),\dots\ (v_{k-1}, v_k)\}$ such that $v_1 = v_j$ and $v_k = v_i$. For each purpose vertex $p \in V^P$ in our graph G, the *reachability subgraph* of p is the graph $G_p = (V_p, E_p)$ where $V_p \subseteq V$ is the set of the vertices that p is reachable from and $E_p \subseteq E$ is the set of edges from G that connect them. If an edge is removed from G, the reachability subgraph of one or more purpose vertices is affected. In general, we write $\mathcal{R}(G_p)$ to denote the set of all subgraphs of the reachability subgraph of p in G - that are still reachability graphs when some edges are removed. Then, to calculate the utility of fulfilling a purpose, for each purpose vertex $p \in V^P$ we define a utility function $u_p : \mathcal{R}(G_p) \to \mathbb{R}_0^+$, which is a function of a reachability subgraph of p.

While u_p can be an arbitrary function dependent on the valuation (or more general, the importance) of datasets in the corresponding reachability subgraph, the valuations of some datasets may influence the valuations of others. To describe the relationships between these valuations in our model, we define a valuation function $\pi : E \to \mathbb{R}_0^+$, representing the valuation of the data propagating through the edge in the data processing system. As we later describe in Section 4, given the reachability subgraph $G_p = (V_p, E_p)$ of a vertex $p \in V^P$, the utility function $u_p(G_p)$ at p can be defined as a function of the valuations of edges in E_p.

In our setting, a user constraint denotes their choice to opt out their data, represented by a user vertex in V^U, from being used for a purpose vertex in V^P. Formally, our set of constraints is a set $\mathcal{N} = \{(v_s, v_t) \mid v_s \in V^U, v_t \in V^P\}$. In order to satisfy the constraints, the initial graph G needs to be modified by removing one or more edges that belong to the paths between pairs (v_s, v_t), such that the utilities u_p are maximised. In essence, our problem is a multi-objective optimisation problem, where the objectives are to maximise u_p, for all $p \in V^P$. The most common approach to multi-objective optimization is to turn the problem into a single-objective optimization using a weighted sum [21]. This allows us to define the utility of G as:

$$U(G) = \sum_{p \in V^P} w_p u_p(G_p), \tag{1}$$

where w_p is the weight of the purpose corresponding to vertex p and G_p is the reachability subgraph of p. Therefore, given $\mathcal{N}$, the CDW problem is to find the consented subgraph of G:

$$G^* = arg \max_{G'} U(G') \tag{2}$$

where $G' = (V, E')$ is a subgraph of G, $E' \subseteq E$, and there is no path from s to t for each $(s, t) \in \mathcal{N}$.

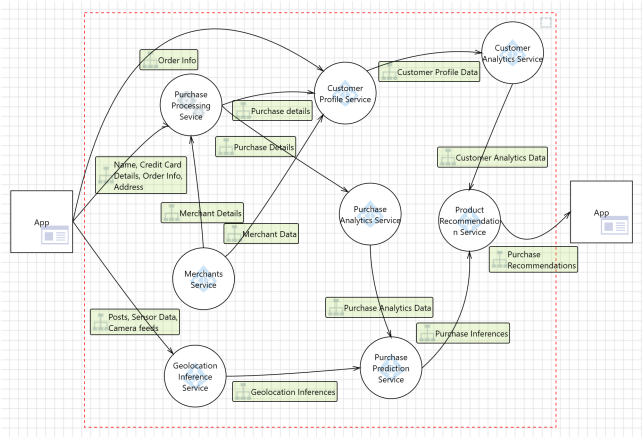


Figure 1: A DFD created with Microsoft Threat Modelling Tool for an example product recommendation feature.

3. RECOMMEND PRODUCT DATA FLOW

Data Flow Diagrams (DFD) [22] are commonly used in the industry to represent data flows in complex systems. For example, Fig. 1 illustrates an example DFD for a single feature: product recommendations. In order to provide product recommendations, user data enters the system though a client application. More specifically, the user's name, credit card details, address and order information are sent to the purchase processing service API. In addition, the order information is sent to the service responsible for constructing customer profiles. Information related to the user's activity such as posts, sensor data or camera feeds are sent to a service producing geolocation-based inferences.

To construct a graph G for this example, firstly, we create vertices V^U for data types collected from the client application: user's name, credit card details, address, order infor-

mation, posts, sensor data and camera feeds. Secondly, we create vertices V^A for all microservices processing the data: the purchase processing service, customer profile service, geolocation inference service, etc. Then, we create a set V^P containing one vertex for the purpose of this feature: serving product recommendations. Finally, we create the edges E connecting vertices in V^U with vertices in V^A the based on the data flow, vertices in V^A with other ones in V^A, and the vertex in V^A representing the service computing product recommendations with the sole purpose vertex in V^P.

An example constraint on data flow of this feature could be a user disagreeing to have their order information used for serving product recommendations. Formally, we would then have a set of constraints $\mathcal{N} = \{(v_s, v_t) \mid v_s \in V^U, v_t \in V^P\}$ where v_s is the vertex corresponding to order information and v_t is our purpose vertex. To solve the problem, we would need to break the data flow from v_s to v_t such that the utility gained from serving product recommendations is maximised. While this example demonstrates a simple use case, in general, a data processing system may consist of many more features on which a user may impose many more constraints.

4. ADDITIVE MODEL

As we prove in [13], in general, CDW is an $\mathcal{NP}$-hard problem. This is because in practice the valuations and utilities defined in Section 2 can be arbitrarily complex functions. In this section, we define a simple but practical instance of the problem, where these functions are linearly additive. In particular, we assume that the valuation function of the data going out of a vertex is linearly additive with respect to the importance of the data on the incoming edges. That is, the value of every node's output is the sum of the values of its inputs. The linear additive model is the simplest model that captures the intuition of each input having an "added value" on the subsequent algorithm. This model may not apply to all settings and, e.g., a sub-additive model might be more appropriate. In many cases the linear additive model could be a reasonable approximation of a model where the inter-dependencies are difficult to measure. Additionally, if we initialise each input edge to have an original value of 1, then the utility of a purpose node, in the this specific setting, sums up how many times the inputs "have been used" in algorithms before reaching the purpose node, thus giving some hint about the overall importance of an input.

In this instance of the problem we consider a DAG $G = (V, E)$ and constraints $\mathcal{N}$, and for each edge $e = (v, v') \in E$, the valuation is defined recursively as follows:

$$\pi(e) = \sum_{e' \in in(v)} \pi(e'). \tag{3}$$

Similarly, we model the utility gained from processing the data for a purpose as a linearly additive function of the data valuations on the incoming edges. That is, for each reachability subgraph G_p, and purpose vertex $p \in V_P$, we define a utility function as follows:

$$u_p(G_p) = \sum_{e \in in(p)} \pi(e). \tag{4}$$

Subsequently we present concrete heuristic implementations for the linear additive case.

5. ALGORITHMS

In this section we focus on a range of algorithms our linearly additive problem. Although there might exist multiple optimal solutions, we design our algorithms looking for a single solution $G^* = (V^*, E^*)$. While some of them offer optimal solutions, others serve as viable heuristics. Note that, even though the algorithms may not be optimal (i.e., utility maximising), all five algorithms always return a *feasible* solution, which is a subgraph of G with no path between each $(s_i, t_i) \in \mathcal{N}$ for $i \in \{1, \ldots, |\mathcal{N}|\}$.

Firstly, a simple heuristic for finding a feasible solution is an algorithm that removes a random edge from each of the paths connecting $(s, t) \in \mathcal{N}$: algorithm RemoveRandomEdge finds all paths from s to t and from each of the paths selects a random edge to remove; then, before the edge is removed, the other edges whose valuation depends on the presence of the given edge in the graph must be updated. In particular, if the valuation of an edge after the update is 0, such edge must also be removed. This solution has a high variance but its run time is polynomial.

Secondly, as the valuation function of the edges is additive, and because the valuation of the incoming edge of an algorithm vertex is always greater or equal than the outgoing one, the removal of the first edge of each path from s to t can serve as another trivial heuristic. Specifically, algorithm RemoveFirstEdge is very similar to RemoveRandomEdge, except that, instead of selecting a random edge, it removes the first edge from each path). This algorithm reflects an approach whereby the user's data is removed entirely and not even collected by the system. Similarly to RemoveRandomEdge, the runtime of this algorithm is polynomial.

Next, we propose a greedy algorithm running in polynomial time. This algorithm follows the heuristic of making locally optimal choices for each constraint. To do so, it uses a polynomial-time algorithm solving the Minimum Cut problem (MinCut) [10, 11] – which is equivalent to the Minimum Multicut when we only have a single constraint $(s, t) \in \mathcal{N}$. Our greedy algorithm RemoveMinCuts, seen in Algorithm 1, first initialises the weights $w(e) = \pi(e) \sum_{p \in r(v)} w_p$ for all edges $e \in E$ (in lines 1 - 4), and then repeatedly calls the MinCut to find the minimum cut that solves MinCut for vertices s and t in $\mathcal{N}$ with weights w. For each edge in the minimum cut, it uses the updateDependencies function to update the valuations of the consecutive edges before removing the given edge. Given that MinCut is known to be solvable in polynomial time [10], the outcome of this heuristic can also be found in polynomial time.

Algorithm 1 RemoveMinCuts

Input: A graph $G = (V, E)$ and a set of constraints $\mathcal{N}$.
Output: A graph G.

1: $w \leftarrow \emptyset$
2: **for all** $e \in E$ **do**
3: $\quad w(e) \leftarrow \pi(e) \sum_{p \in r(v)} w_p$
4: **for all** $(s, t) \in \mathcal{N}$ **do**
5: $\quad$ **for all** $e \in$ MinCut(G, w, s, t) **do**
6: $\quad\quad$ **if** `hasEdge`(G, e) **then**
7: $\quad\quad\quad$ `updateDependencies`(G, e)
8: $\quad\quad\quad$ `removeEdge`(G, e)

Another way of approximating the solution is by converting our problem to MinMC. That is, we can solve MinMC with weights $w(e) = \pi(e) \sum_{p \in r(v)} w_p$ for all edges $e \in E$ and then use the MinMC solution to find a solution. In the same way as RemoveMinCuts, shown in Algorithm 2, RemoveMinMC starts from initialising the weights w. Then, it finds the minimum multicut of graph G for constraints $\mathcal{N}$. Subsequently, for each edge in the minimum multicut, it uses the updateDependencies function to update the valuations of the consecutive edges (in line 8) before removing the given edge.

Algorithm 2 RemoveMinMC

Input: A graph $G = (V, E)$, a set of constraints $\mathcal{N}$.
Output: A graph G.

```
1: w ← ∅
2: for all e ∈ E do
3:     w(e) ← π(e) Σ_{p∈r(v)} w_p
4: multicut ← MinMC(G, N, w)
5: for all e ∈ multicut do
6:     if hasEdge(G, e) then
7:         updateDependencies(G, e)
8:         removeEdge(G, e)
```

Finally, we propose an algorithm that can guarantee achieving an optimal solution. In Algorithm 3, BruteForce is an exhaustive search algorithm that enumerates all feasible candidates for the solution and compares them to eventually output the one that maximises the utility. More specifically, BruteForce starts from finding the set of all paths $\mathcal{A}$ from s to t for all $(s, t) \in \mathcal{N}$, which need to be broken. In order to list all feasible multicuts of G for the given $\mathcal{N}$, the Cartesian product of $\mathcal{A}$ is computed (in line 5). Then, the algorithm systematically checks the utility of G after the removal of each multicut. Importantly, at the beginning of the multicut check, copies are made of the valuation values π' of each edge and the number of paths p' the edge belongs to in G, as well as of the graph G itself. Before an edge of the feasible multicut is removed from the copy of G, the valuation π' and the number of paths p' in the copy of G are updated for its dependencies. At the end of the multicut check, the utility of the copy of G is compared to the utility of the most optimal solution found so far. This way the algorithm can guarantee finding the optimal solution but is exponential even in the best case.

Notably, all of the presented algorithms generalize beyond the linearly additive model. What is specific to this particular model in RemoveMinCuts, RemoveMinMC and BruteForce is the way the weights $w(e)$ are assigned and the dependencies are updated, which depends on the valuation function $\pi(e)$ and weights w_p.

6. OPTIMALITY OF SOLUTIONS

Out of the five algorithms, only BruteForce guarantees an optimal solution. In contrast, it is clear that RemoveRandomEdge does not guarantee an optimal solution – we use it as a benchmark for our evaluation. Let us analyse the properties of the solutions returned by our three remaining heuristics and prove that none of them can guarantee finding an optimal solution even for the linear setting.

Algorithm 3 BruteForce

Input: A graph $G = (V, E)$ and a set of constraints $\mathcal{N}$.
Output: A graph G^*.

```
1: A ← ∅
2: for all (s, t) ∈ N do
3:     A ← A ∪ getAllEdgePaths(G, s, t)
4: multicuts ← cartesianProduct(A)
5: maxUtility ← 0
6: G* ← G
7: for all multicut ∈ multicuts do
8:     G' ← G
9:     π', p ← ∅, ∅
10:    for all e ∈ E do
11:        π'(e) ← π(e)
12:        p(e) ← Σ_{p∈r(v)} w_p
13:    for all e ∈ multicut do
14:        if hasEdge(G', e) then
15:            updateDependencies(G', e, π', p)
16:            removeEdge(G', e)
17:    utility ← U(G')
18:    if utility > maxUtility then
19:        maxUtility ← utility
20:        G* ← G'
21: return G*
```

Firstly, we show that a simple removal of the first edge of each path by the RemoveFirstEdge does not guarantee an optimal solution, i.e., for each $(s_i, t_i) \in \mathcal{N}$, there is at least one path $P = (V_P, E_P) \in \mathcal{A}$ of the form $V_P = \{v_1, v_2, \ldots, v_k\}$, $E_P = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)\}$ where $v_1 = s_i$ and $v_k = t_i$. From each such path $P \in \mathcal{A}$, we could remove edge (v_1, v_2). We refer to (v_1, v_2) as the *first edge*. Let G be a data processing model where $V^U = \{v_1\}$, $V^A = \{v_2\}$, $V^P = \{v_3, v_4\}$, $E = \{(v_1, v_2), (v_2, v_3), (v_2, v_4)\}$ and for each $p \in V^P$, $w_p = 1$. In addition, assume that for edge $e_1 = (v_1, v_2)$, $\pi(e_1) = a$ where $a \in \mathbb{R}_0^+$, and that $\mathcal{N} = \{(v_1, v_3)\}$.

In such case, we use Eq. 3 to calculate the valuation of edges $e_2 = (v_2, v_3)$ and $e_3 = (v_2, v_4)$, which is $\pi(e_2) = \pi(e_3) = a$. We also use Eq. 1 to calculate the initial utility of G, which is $U(G) = 2a$. Given that $\mathcal{N} = \{(v_1, v_3)\}$, we establish that there is one path that needs to be disconnected in order to satisfy the constraints, i.e. $\mathcal{A} = \{P\}$ where $P = (\{v_1, v_2, v_3\}, \{(v_1, v_2), (v_2, v_3)\})$.

Then, we remove the first edge (v_1, v_2) from P. The utility of the resulting graph G_1' is $U(G_1') = 0$, since purpose vertices v_3 and v_4 are now not linked to any user vertex. However, if we instead removed the edge (v_2, v_3), vertex v_4 would still be linked to v_1 and therefore the utility of the resulting graph G_2' would be $U(G_2') = a$. Thus, the removal of the first edge does not provide us with an optimal solution.

Furthermore, RemoveMinCuts does not lead to an optimal solution. Consider a series of graphs $G^0, G^1, \ldots, G^{|\mathcal{N}|}$. These graphs are computed recursively such that $G^0 = G = (V, E)$ and for all $i \in \{1, \ldots, |\mathcal{N}|\}$, $G^i = (V, E^i)$ where $E^i = E^{i-1} \setminus \text{MinCut}(G^{i-1}, s_i, t_i, w)$ corresponds to i-th pair of vertices in $\mathcal{N}$. Given a solution to MinCut, one could transform the problem into a repeated MinCut problem looking for $G^{|\mathcal{N}|}$. While this approach leads to a feasible so-

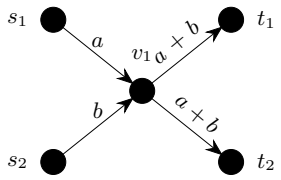


Figure 2: A data processing model where $V^U = \{s_1, s_2\}$, $V^A = \{v_1\}$, $V^P = \{t_1, t_2\}$ and $E = \{(s_1, v_1), (s_2, v_1), (v_1, t_1), (v_1, t_2)\}$.

lution, we show that $G^{|\mathcal{N}|}$ is not an optimal solution. Let G be a graph where $V^U = \{s_1, s_2\}$, $V^A = \{v_1\}$, $V^P = \{t_1, t_2\}$, $E = \{(s_1, v_1), (s_2, v_1), (v_1, t_1), (v_1, t_2)\}$ and for each $p \in V^P$, $w_p = 1$. Assume that for $e_1 = (s_1, v_1)$, $\pi(e_1) = a$ and for $e_2 = (s_2, v_1)$, $\pi(e_2) = b$, where $a, b \in \mathbb{R}_0^+$ and $a > b$ (see Fig. 2). In addition, $\mathcal{N} = \{(s_1, t_1), (s_1, t_2)\}$. We look for $G^2 = (V, E \setminus \textsc{MinCut}(G^1, s_1, t_1, w))$. Thus, we first calculate $G^1 = (V, E \setminus \textsc{MinCut}(G, s_1, t_1, w))$. We observe that there is one path $P = (\{s_1, v_1, t_1\}, \{(s_1, v_1), (v_1, t_1)\})$ between vertices s_1 and t_1. Because $a > b$, $w((v_1, t_1)) = a + b < w((s_1, v_1)) = 2a$. Thus, $G^1 = (V, E \setminus \{(v_1, t_1)\})$. With this information, we return to looking for G^2. We observe that there is one path $P = (\{s_1, v_1, t_2\}, \{(s_1, v_1), (v_1, t_2)\})$ between vertices s_1 and t_2. However, now $w((s_1, v_1)) = a$ and $w((v_1, t_2)) = a + b$. So, $w((s_1, v_1)) < w((v_1, t_2))$ and $G^2 = (V, E \setminus \{(s_1, v_1), (v_1, t_1)\})$. After that, we calculate $U(G^2) = b$. However, we can see that to obtain an optimal solution, it is sufficient to remove (s_1, v_1) only: the optimal solution is $G^* = (V, E \setminus \{(s_1, v_1)\})$, because its utility is $U(G^*) = 2b$. Thus, G^2 is not an optimal solution.

Intuitively, it is reasonable to assume that the optimal solution requires removing no more than one edge per path between s and t for each $(s, t) \in \mathcal{N}$. Let $T \subseteq V^P$ be a set of purpose vertices such that for all $(s, t) \in \mathcal{N}$, $t \in T$. If for each path $P = (V_P, E_P) \in \mathcal{A}$ there is exactly one edge $e \in E_P$ such that $e \in E_{MinMC}$, then the removal of E_{MinMC} reduces the utility of a purpose vertex $t \in T$ by $\sum_{e \in E_t} \pi(e)$ where $E_t \subseteq E_{MinMC}$ is a set of those edges in E_{MinMC} that are within the reachability subgraph of t, G_t. Thus, if the resulting graph after the removal of set E_{MinMC} from G is $G' = (V, E')$, then using Eq. 1, the total loss of utility is:

$$U(G) - U(G') = \sum_{t \in T} \sum_{e \in E_t} w_t \pi(e). \tag{5}$$

This is equivalent to the following equation:

$$U(G') = U(G) - \sum_{e \in E \setminus E'} \pi(e) \sum_{t \in T} w_t. \tag{6}$$

In fact, if we plug Eq. 6 into Equation 2, we have:

$$G^* = arg \max_{G'} \{U(G) - \sum_{e \in E \setminus E'} \pi(e) \sum_{t \in T} w_t\}. \tag{7}$$

Equivalently, we are looking for a subgraph where:

$$E^* = E \setminus \{arg \min_{E \setminus E'} \sum_{e \in E \setminus E'} \pi(e) \sum_{t \in T} w_t\}. \tag{8}$$

Thus, E^* is the set difference of E and the minimum multicut of G given $\mathcal{N}$, where the edge weight is $w(e) = \pi(e) \sum_{t \in T} w_t$. In more detail, the minimum multicut with edge weights $w(e) = \pi(e) \sum_{t \in T} w_t$ can be expressed as:

$$E_{MinMC} = arg \min_{E'} \sum_{e \in E'} \pi(e) \sum_{t \in T} w_t. \tag{9}$$

If we plug Eq. 8 into Eq. 9, then what we are looking for is $G^* = (V, E^*)$ where:

$$E^* = E \setminus E_{MinMC}. \tag{10}$$

Since E_{MinMC} is a solution to MinMC, there is $G^* = (V, E \setminus E_{MinMC})$.

However, removing a single edge from each path does not always result in an optimal solution. Consider a graph G where $V^U = \{s_1, s_2\}$, $V^A = \{v_1\}$, $V^P = \{t_1, t_2\}$, $E = \{(s_1, v_1), (s_2, v_1), (v_1, t_1), (v_1, t_2)\}$ and for each $p \in V^P$, $w_p = 1$. In addition, assume that for $e_1 = (s_1, v_1)$, $\pi(e_1) = a$ and for $e_2 = (s_2, v_1)$, $\pi(e_2) = b$, where $a, b \in \mathbb{R}_0^+$ and $a > b$ (see Fig. 2). Now, let the set of constraints be as follows: $\mathcal{N} = \{(s_1, t_1), (s_1, t_2), (s_2, t_1)\}$. We can see that the optimal solution in this case is $G^* = (V, \{(s_2, v_1), (v_1, t_2)\})$. However, since $(s_1, t_1) \in \mathcal{N}$ and there is a path from s_1 to t_1 in the original graph G, we can observe that the optimal solution G^* does not contain two edges (s_1, v_1) and (v_1, t_2) from that path.Therefore, in general, it is not true that RemoveMinMC can guarantee finding an optimal solution.

7. EXPERIMENTAL EVALUATION

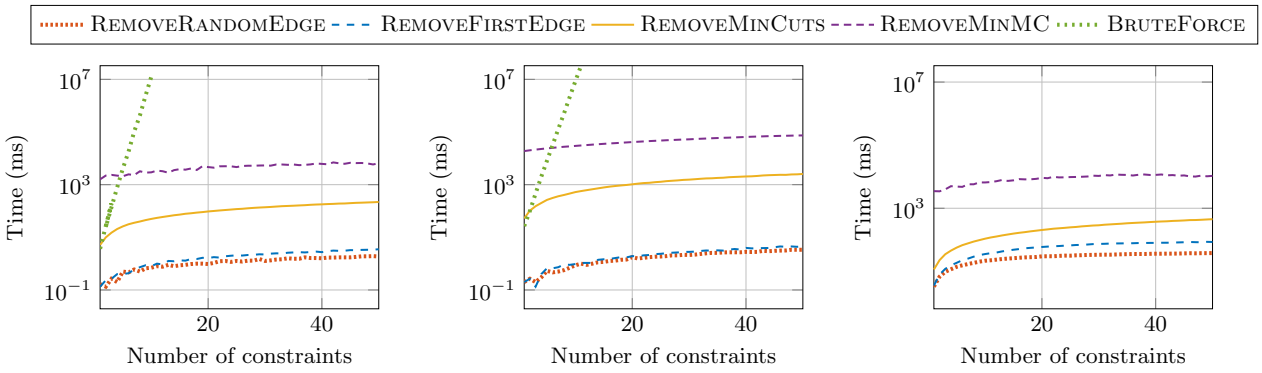
To evaluate the algorithms' performance empirically, we perform experiments on the University of Southampton High Performance Computing service *Iridis 4*, measuring their performance on synthetic data. Our graph generation method includes the following parameters:

- number of constraints $|\mathcal{N}|$;
- number of vertices $|V|$;
- path length k – for any $(s, t) \in \mathcal{N}$, if there is a path $P = \{(v_1, v_2) \dots (v_{k-1}, v_k))\}$ such that $v_1 = s$ and $v_k = t$, then k defines the number of workflow stages, i.e. data that 'flows' from s to t though $k-2$ algorithm nodes;
- vertex distribution vector X_k – proportions of vertices at different data flow stages, e.g. a setting $X_k = (50\%, 25\%, 10\%, 10\%, 5\%)$ represents a scenario for $k = 5$ where half of the vertices are the user data vertices, 45% are the algorithm vertices and 5% are the number of the purpose vertices (NU - non-uniform distribution, U - uniform distribution);
- minimum density d – the proportion of initially generated edges between any two workflow stages.

We prepare three datasets with different configurations of the above parameters, specified in Tab. 1.

We measure the runtime of the algorithms on 100-vertex graphs (dataset 1a), on 10 times larger, 1000-vertex graphs (dataset 1b) and on 100-vertex graphs where the number of edges between each level of data processing is at least 20% of all possible edges (dataset 1c). We observe that the runtime

The Iridis Compute Cluster, `https://cmg.soton.ac.uk/iridis`.



(a) 100-vertex graphs (dataset 1a). (b) 1000-vertex graphs (dataset 1b). (c) Dense graphs (dataset 1c).

Figure 3: The number of constraints vs. the runtime of the algorithms in graphs from dataset 1.

Table 1: Parameter configurations for datasets 1, 2 and 3.

	Dataset 1			**Dataset 2**	**Dataset 3**
	a	**b**	**c**		
$\lvert\mathcal{N}\rvert$	1 – 50	1 – 50	1 – 50	10	5
$\lvert V\rvert$	100	1000	100	150 – 5000	100 – 10000
k	5	5	5	3 – 50	5
X_k	NU	NU	U	U	NU
d	0	0	20%	0	0

Table 2: Comparison of the graph's utility after applying RemoveMinMC and BruteForce.

Number of constraints	RemoveMinMC		BruteForce	
	% of original	SE	% of original	SE
1	97.79	0.32	97.79	0.32
2	95.08	0.35	95.08	0.35
3	92.71	0.58	92.71	0.58
4	90.62	0.57	90.63	0.57
5	88.59	0.75	88.65	0.75
6	86.59	0.72	86.66	0.72
7	84.71	0.72	84.77	0.71
8	83.22	0.70	83.28	0.70
9	81.24	0.69	81.33	0.69
10	79.30	0.69	79.39	0.68

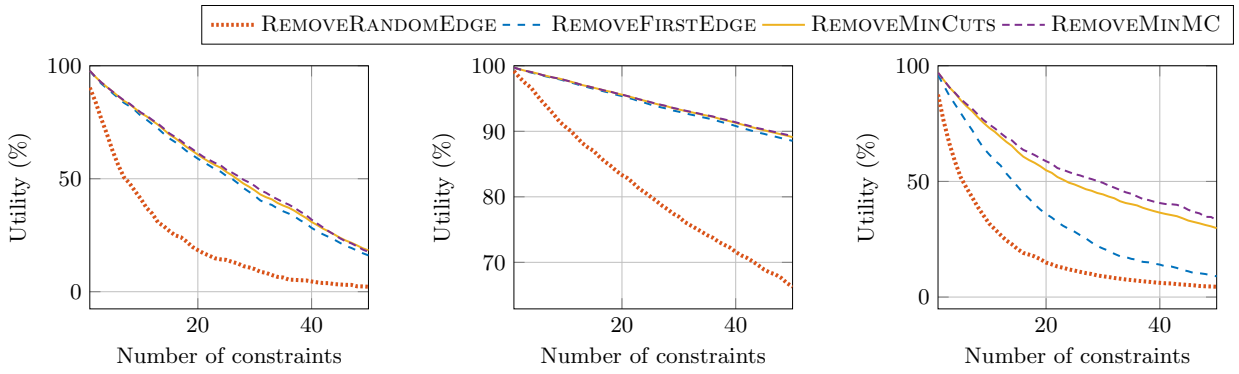
of BruteForce increases rapidly with the increasing number of constraints, reaching an average time of 14838508.46 ms (i.e. over 4 h) for just 10 constraints on dataset 1a and 8563968 ms (i.e. over 2 h) on dataset 1b. For dataset 1c, in most cases, BruteForce is unable to return a result in 60 hours even for just a single pair of constraints. Thus, although BruteForce guarantees finding an optimal solution, its average runtime makes this algorithm impractical. At the same time, the solver-based RemoveMinMC can reach an approximate solution for even 50 constraints on average in 6.51s on dataset 1a, 74.1s on dataset 1b and 11s on dataset 1c. RemoveMinCuts can on average find an approximate solution for 50 constraints in 220.2 ms on dataset 1a, 2.55s on dataset 1b and 450.57 ms on dataset 1c.

Moreover, we observe a change in the graph's utility as the number of constraints grows. Although the exponential runtime of the BruteForce algorithm means we cannot run the experiments for more than 10 constraints, we still compare the results to RemoveMinMC for this limited setting. Results are presented in Tab.2 and show that the utility using RemoveMinMC is nearly optimal in this case. Although the algorithm is only guaranteed to be optimal for specific settings, this empirical outcome suggests that, for graphs with a relatively small number of constraints, RemoveMinMC is likely to provide very accurate solutions. Moreover, Fig. 4, shows that the differences in utility between algorithms is more evident when the graphs are denser, resulting in significantly poorer performance of RemoveMinCuts, RemoveFirstEdge and RemoveRandomEdge. Nonetheless, RemoveMinMC provides the best solution for all three types of graphs.

Next, we observe how the execution time depends on the number of paths between pairs of vertices that connect the constraints. Fig. 5 presents a scatter plot of the runtime of the algorithms and distribution of utility for dataset 1c. In particular, we can see that, in case of RemoveMinCuts and RemoveMinMC, the runtimes increase almost linearly with respect to the number of paths. However, the execution times for these two algorithms differ significantly. For example, for a graph where 822 paths are required to be broken, RemoveMinMC takes 12116 ms to return a solution, whereas RemoveMinCuts can provide one in only 472 ms. Similarly, we can see that the utility decreases as the number of paths connecting constraints increases. Yet again, the utility after executing RemoveMinMC tends to decrease the slowest, reaching on average the utility of 32.27% of the original utility for the graph with 822 paths to be broken. For the same graph, the next best solution is RemoveMinCuts with an accuracy of 24.58%. For comparison, RemoveFirstEdge achieves on average utility of 15.73%.

Next, we apply the algorithms to graphs with a constant number of paths. Since only edges connected to purpose vertices affect the utility, increasing the lengths of the paths on its own does not affect the utility. Thus, we focus on the algorithm runtime as the path length grows. In Fig. 6, we consider sparse graphs with vertices distributed uniformly with the same number of user data vertices as purpose vertices (dataset 2). We can see that as the path length grows, the runtime in case of BruteForce increases faster.

Lastly, we analyse how the number of vertices in the graph impacts the runtime and the utility of the graph after ap-



(a) 100-vertex graphs (dataset 1a). (b) 1000-vertex graphs (dataset 1b). (c) Dense graphs (dataset 1c).

Figure 4: The number of constraints vs. graph utility after applying the algorithms on graphs from dataset 1.

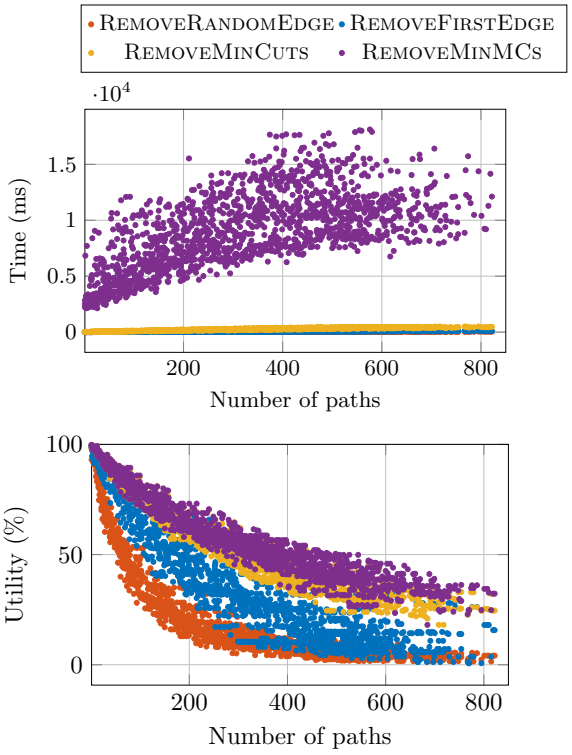


Figure 5: No. of paths vs. runtime and utility (dataset 1c).

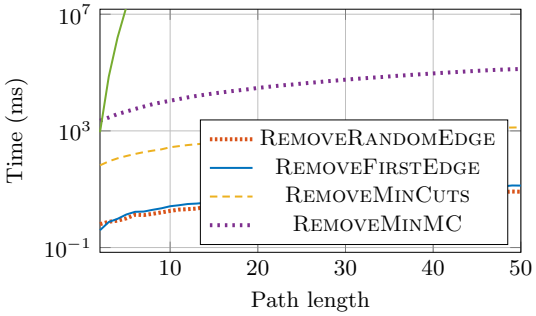


Figure 6: Path length vs. time in sparse graphs (dataset 2).

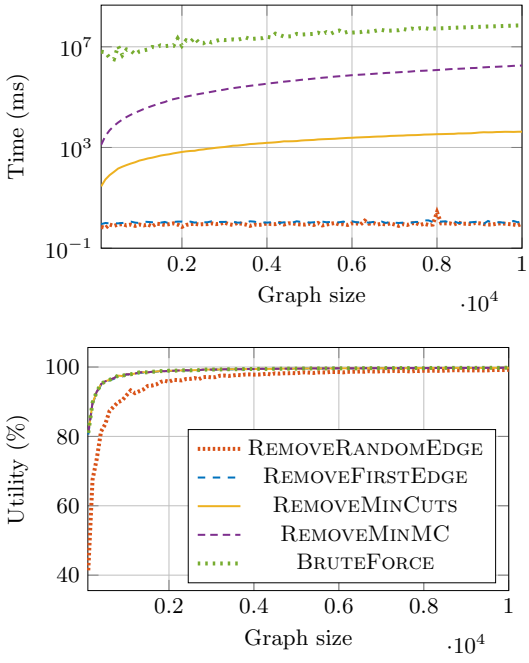


Figure 7: Graph size vs. runtime and utility (dataset 3).

plying the algorithms. To do this, we run the algorithms on graphs from dataset 3. As the number of paths between the constraints and their length are equal for these graphs, in Fig. 7 we can see that the size of the graph has only a slight impact on the execution time for BruteForce. In addition, RemoveMinCuts is faster on average compared to BruteForce and RemoveMinMC. Considering the utility, Fig. 7 shows that the graph size does not have significant impact on utility when the number of paths between the constraints and their length remain equal for the graphs.

8. OPEN PROBLEMS

In this paper, we designed a theoretical mechanism where the data flow is structured as a graph and privacy constraints collected from the user point to pairs of vertices in this graph. While our approach is effective and finds a nearly optimal solution for large graphs, it also creates a range of open problems and challenges.

First, for our algorithms to deliver value, a service provider needs to build an accurate graph. In particular, for large-scale systems with thousands of microservices built on billions lines of code in multiple languages, identifying data flows and all the purposes they are processed for, is challenging and time-consuming. Therefore, future work should develop automated tools to support graph generation.

Second, some of our effective algorithms rely on the simplifying assumption that the value of different information sources is additive. While this is a reasonable starting point and can be a reasonable approximation in some settings, in practice the value may be subadditive (e.g. in the case of redundant data) or superadditive (when data complements each other). As data processing systems keep expanding, future work should focus on exploring more variable models that align the utility and valuation functions with real-world use cases. While the algorithms proposed in this article conceptually generalize to more complex settings, models with different utility and valuation functions may open opportunities for designing more efficient algorithms. For example, in certain machine learning as well as statistical algorithms there is already work that addresses masking or distorting the input with noise. This gives rise to an exciting generalisation of our framework, where utility is affected not by removing the input edge but by distorting the input, and in the future we plan to explore this.

Third, we show that the problem in general is NP-hard. This result provides us with the lower bound on the complexity of the problem. The upper bound, however, depends largely on the complexity of the selected valuation and utility functions. In the future, we plane to investigate the upper bound of the problem in different settings.

In addition, there are several open problems regarding the scalability of the problem. Currently, our solution needs to be recomputed every time a new user enters the system or when an existing user updates their constraints. What is more, every time a change is made, some of the algorithms that process the data would need to be re-run as well, which could be costly. At the same time, there could be many users of the same type, i.e. with similar privacy constraints, and a limited number of different user types which can be known in advance. To take advantage of this, users of the same type could, e.g., be treated as a single user to cope with thousands and even millions of users. This way, if new users enter the system, a new solution can be found quickly. Generally, as more and more users have privacy constraints, new methods are needed that take into account scalability by re-using some of the computation performed for the previous solutions, as well as the costs of making changes.

Finally, there are plenty of opportunities to consider richer types of privacy constraints and user preferences. For example, users may have constraints on combinations of different data types for a specific purpose (e.g. a user may say '*I'm okay with you using my data for advertising, but don't combine my location with my purchase history*') or time restrictions on data processing (*'I'm okay with you sharing my purchase history, but I don't want them to keep it for more than 30 days*'). Future work should formulate new problems around these constraints, as well as constraint the secondary re-use of data later in the data lifecycle.

9. RELATED WORK

With the scale of data collection and processing growing vastly in the recent years, there has been an emergence of initiatives and tools that aim to aid users in controlling the flow of their personal data. Early efforts include the Platform for Privacy Preferences (P3P) which aimed to enable machine-readable privacy policies [7]. Such privacy policies could be automatically retrieved by Web browsers and other tools that can display symbols, prompt users, or take other appropriate actions. Users were able to communicate their privacy constraints to these so-called *user agents* as a list of rules expressed in a P3P Preference Exchange Language (APPEL) [7]. The agents were then able to compare each policy against the user's constraints and assist the user in deciding when to exchange data with websites [7].

However, the P3P lacked a mechanism that would allow for enforcement of the privacy policy within the enterprise and for management of users' individual privacy preferences [6, 3, 17]. Thus, a new approach was proposed [6] where service providers would publish privacy policies, collect and manage user preferences and consent, and enforce the policies throughout their systems. Based on this framework, the Platform for Enterprise Privacy Practices (E-P3P) was developed [5, 16]. Our work builds on this idea and proposes how to satisfy the users' individual preferences *optimally*.

Nonetheless, more general approaches that go beyond the P3P were required to address the need for policy enforcement. To that end, Hippocratic databases were proposed [3, 4, 1, 19, 2] as 'database systems that take responsibility for the privacy of data they manage'. There, personal data was associated with the purposes it was collected for, and metadata such as retention period. Given privacy constraints, the so-called Privacy Constraint Validator would check if the policy is acceptable to the user. Recently, an approach to enable even more fine-grained control of such policies within relational databases was presented [18]. Here, we extend this approach.

Complementing our work, related work has proposed methods for privacy policy-compliant data processing. For example, once it is known how the user's privacy constraints can be optimally satisfied (e.g., using our proposed approach), it is possible to ensure that the datasets used by the data-processing algorithms align with these constraints [8, 9]. Moreover, data processing techniques can be selected based on the consented types of the input data [24]. Furthermore, to make sure that the policies are enforced, a system for checking data usage policies automatically at query runtime has been proposed [23].

10. ACKNOWLEDGEMENTS

E. Gerding was partially funded by the EPSRC-funded platform grant "AutoTrust: Designing a Human-Centred Trusted, Secure, Intelligent and Usable Internet of Vehicles" (EP/R029563/1). G. Konstantinidis was partially funded by the UKRI Horizon Europe guarantee funding scheme for the Horizon Europe projects RAISE (No. 101058479) and UPCAST (No. 101093216).

11. REFERENCES

[1] R. Agrawal, R. Bayardo, C. Faloutsos, J. Kiernan, R. Rantzau, and R. Srikant. Auditing compliance with a hippocratic database. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 516–527, 2004.

[2] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, and W. Rjaibi. Extending relational database systems to automatically enforce privacy policies. In *21st International Conference on Data Engineering (ICDE'05)*, pages 1013–1022. IEEE, 2005.

[3] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *Proceedings of the 28th VLDB Conference*, pages 143–154, United States, 2002. VLDB Endownment.

[4] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Implementing p3p using database technology. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*, pages 595–606. IEEE, 2003.

[5] P. Ashley, S. Hada, G. Karjoth, and M. Schunter. E-p3p privacy policies and privacy authorization. In *Proceedings of the 2002 ACM Workshop on Privacy in the Electronic Society*, pages 103–109. ACM, 2002.

[6] P. Ashley, C. Powers, and M. Schunter. From privacy promises to privacy management: a new approach for enforcing privacy throughout an enterprise. In *Proceedings of the 2002 Workshop on New Security Paradigms*, pages 43–50. ACM, 2002.

[7] L. F. Cranor. *Web privacy with P3P*. O'Reilly Media, Inc., 2002.

[8] C. Debruyne, H. J. Pandit, D. Lewis, and D. O'Sullivan. Towards generating policy-compliant datasets. In *2019 IEEE 13th International Conference on Semantic Computing (ICSC)*, pages 199–203. IEEE, 2019.

[9] C. Debruyne, H. J. Pandit, D. Lewis, and D. O'Sullivan. "just-in-time" generation of datasets by considering structured representations of given consent for gdpr compliance. *Knowledge and Information Systems*, 62(9):3615–3640, 2020.

[10] Y. Dinitz. Dinitz'algorithm: The original version and Even's version. In O. Goldreich, A. L. Rosenberg, and A. L. Selman, editors, *Theoretical Computer Science: Essays in Memory of Shimon Even*, pages 218–240. Springer, 2006.

[11] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[12] European Parliament and the Council. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union*, pages 1–88, May 2016.

[13] D. Filipczuk, E. H. Gerding, and G. Konstantinidis. Consent management in data workflows: A graph problem. In *Proceedings of the 26th International Conference on Extending Database Technology (EDBT)*, 2023.

[14] A. Ghorbani and J. Zou. Data Shapley: Equitable valuation of data for machine learning. In *International Conference on Machine Learning*, pages 2242–2251, Long Beach, California, USA, 2019. PMLR.

[15] D. Huye, Y. Shkuro, and R. R. Sambasivan. Lifting the veil on {Meta's} microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 419–432, 2023.

[16] G. Karjoth, M. Schunter, and M. Waidner. Platform for enterprise privacy practices: Privacy-enabled management of customer data. In *2nd Workshop on Privacy-Enhancing Technologies. Lecture Notes in Computer Science*, pages 69–84. Springer, 2002.

[17] J. H. Kaufman, S. Edlund, D. A. Ford, and C. Powers. The social contract core. In *Proceedings of the 11th International World Wide Web Conference (WWW)*, pages 210–220, Honolulu, Hawaii, 2002. ACM.

[18] G. Konstantinidis, J. Holt, and A. Chapman. Enabling personal consent in databases. *Proceedings of the VLDB Endowment*, 15(2):375–387, October 2021.

[19] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt. Limiting disclosure in Hippocratic databases. In *Proceedings of the 30th International Conference on Very Large Databases*, pages 108–119. VLDB Endowment, 2004.

[20] C. Li, D. Y. Li, G. Miklau, and D. Suciu. A theory of pricing private data. *ACM Transactions on Database Systems*, 39(4):1–28, 2014.

[21] R. T. Marler and J. S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, 2004.

[22] L. Sion, K. Yskout, D. Van Landuyt, and W. Joosen. Solution-aware data flow diagrams for security threat modeling. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1425–1432, 2018.

[23] P. Upadhyaya, M. Balazinska, and D. Suciu. Automatic enforcement of data use policies with DataLawyer. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 213–225. ACM, 2015.

[24] Y. Wang and A. Kobsa. Respecting users' individual privacy constraints in web personalization. In *International Conference on User Modeling*, pages 157–166. Springer, 2007.