General Data Protection Runtime: Enforcing Transparent GDPR Compliance for Existing Applications

David Klein
Technische Universität Braunschweig
david.klein@tu-braunschweig.de

Benny Rolle
SAP SE
benny.rolle@sap.com

Thomas Barber
SAP Security Research
thomas.barber@sap.com

Manuel Karl
Technische Universität Braunschweig
m.karl@tu-braunschweig.de

Martin Johns
Technische Universität Braunschweig
m.johns@tu-braunschweig.de

ABSTRACT

Recent advances in data protection regulations brings privacy benefits for website users, but also comes at a cost for operators. Retrofitting the privacy requirements of laws such as the General Data Protection Regulation (GDPR) onto legacy software requires significant auditing and development effort. In this work we demonstrate that this effort can be minimized by viewing data protection requirements through the lens of information flow tracking. Instead of manual inspections of applications, we propose a lightweight enforcement engine which can reliably prevent unlawful data processing even in the presence of bugs or misconfigured software.
Taking GDPR regulations as a starting point, we define twelve software requirements which, if implemented properly, ensure adequate handling of personal data. We go on to show how these requirements can be fulfilled by proposing a metadata structure and enforcement policies for dynamic information flow tracking frameworks. To put this idea into practice, we present Fontus, a Java Virtual Machine (JVM) information flow tracking framework, which can transparently label personal data in existing Java applications in order to aid compliance with data protection regulations. Finally, we demonstrate the applicability of our approach by enforcing data protection polices across 7 large, open source web applications, with no changes required to the applications themselves.

CCS CONCEPTS

• **Security and privacy → Information flow control**;

KEYWORDS

GDPR Enforcement, Taint-Tracking, Data Protection, Privacy

ACM Reference Format:
David Klein, Benny Rolle, Thomas Barber, Manuel Karl, and Martin Johns. 2023. General Data Protection Runtime: Enforcing Transparent GDPR Compliance for Existing Applications. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3616604

CCS '23, November 26–30, 2023, Copenhagen, Denmark
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3616604

1 INTRODUCTION

The ubiquity of web applications in modern life has led to increasing tensions between their operators and users. While companies have seen the value in collecting user's data for marketing or advertising purposes, the users themselves want to restrict such data collection to protect their own privacy. These concerns have been enshrined in legislation, with data protection laws imposing strict restrictions on the collection and processing of personal data, with severe penalties for non-compliance. In particular, the European Union's General Data Protection Regulation [9] (GDPR) can impose penalties of up to 20 million Euro, or up to 4% of the total worldwide annual turnover, whichever is higher, leading to significant fines [e.g., 4, 34, 35]. Companies therefore have a strong financial incentive to process personal data in a compliant manner, and may even consider offering increased privacy protection as a competitive feature.

However, enforcing data protection policies such as the GDPR poses a significant challenge for website developers and operators. For example, consider a modern enterprise web application utilizing the Java Spring Boot framework [51] to implement an e-commerce application. When purchasing items, a user provides personal information such as their e-mail and postal address in order to purchase goods on the site. During this process, their personal data will be processed by multiple components in the Spring stack: from the HTML form in the user's browser, via Java code in the Spring back-end, to storage in an SQL database. Tracking the usage of personal data in even the simplest web applications quickly becomes a near-impossible task, especially considering that even the Spring PetClinic reference application [47] loads over 11,200 classes and requires downloading of 113 packages from the Maven Central repository. This difficulty is confirmed by recent studies [37], which find 751 GDPR fines issued since 2018 due to unauthorized data processing. Spiekermann accurately summarizes: "Data is like water: it flows and ripples in ways that are difficult to predict" [44, p. 38].

A number of mechanisms have been proposed to aid developers create applications which are compliant with the GDPR. For example, Ferrara et al. [14] propose a technique to statically detect

GDPR violations, but do not allow dynamic evaluation of compliance based on an individual user's consent. Mehta et al. [31] consider attaching policies to database entries, but do not consider the rest of the application stack. On the other hand, Wang et al. [52] propose a system which ensures lawful data processing across an entire application, but rely on the presence of dedicated trusted hardware.

Instead, we present a framework for web applications which can dynamically enforce GDPR policies based on an individual user's consent. Our approach leverages non-intrusive dynamic information-flow tracking in the form of tainting and therefore requires no modification of (or even access to) an application's source code, while preventing GDPR violations even in complex applications with a large number of rapidly changing dependencies. We propose transparently attaching a user's consent preferences and further metadata to personal data as it enters a web application. By tracking the propagation of the labels during program execution, we can enforce restrictions on the processing of data which would otherwise violate the GDPR, for example, due to bugs or misconfiguration of the software. At the same time, our framework can assist operators to perform other activities related to the GDPR, such as maintaining a record of processing activities or data breach notifications. To summarize, our contributions are the following:

- Twelve software requirements which, when implemented correctly, can help to prevent GDPR violations.
- Metadata structures and propagation rules for information flow tracking which fulfill said requirements, and can be used to automate GDPR policy enforcement.
- A dynamic taint tracking engine for Java applications with support for persisting taint information into SQL databases.
- Integration of the GDPR policy enforcement concept into our taint engine, demonstrating its feasibility on three major open source Java web applications.

The remainder of this paper is structured as follows: we first introduce the required legal and technical background information (Section 2) and derive twelve software requirements enabling lawful processing of personal data under the GDPR (Section 3). We then map these requirements onto a metadata structure for dynamic data-flow tracking (Section 4). We go on to show how this concept can be realized in practice for modern web applications (Section 5). Afterwards we present Fontus, our prototype implementation which integrates support for such data protection enforcement into a taint tracking engine for the Java Virtual Machine (JVM) (Section 6). We continue by demonstrating the practical viability of our approach (Section 7) with 7 large, open source web applications. Lastly, we discuss our findings and challenges (Section 8), related work (Section 9) and finally conclude in Section 10.

We also prepared a supplemental document with additional information about the performance evaluation. It is available at https://github.com/ias-tubs/gdpr_tainting.

2 BACKGROUND

First, we give a broad overview of the GDPR which governs the processing of personal data within the EU. Secondly, we explain the fundamental concepts of dynamic information flow tracking techniques which we build on later in the paper. Unless otherwise indicated, references to Articles (Art.) and Recitals (Rec.) below refer to the GDPR.

2.1 Personal Data Processing under GDPR

The central term in data protection law is *personal data*: The Charter of Fundamental Rights of the European Union (CFR) states in Art. 8(1) that "Everyone has the right to the protection of personal data concerning him or her". Art. 4(1) GDPR defines "personal data" as "any information relating to an identified or identifiable natural person ('data subject')." In the EU, the principle of prohibition with reservation of permission applies to the processing of personal data (Art. 6(1) GDPR, Art. 8(2) CFR), meaning that personal data can only be processed if backed up with a legal basis. In addition, for the lawful processing of personal data, further principles must be complied with. These include, among others:

- Purpose limitation: The purposes for which the data are processed must be determined prior to processing (Art. 5(1)(b)).
- Data minimization: The amount of processed personal data must be limited to what is necessary for the respective purpose (Art. 5(1)(c)).
- Storage limitation: Personal data must not be stored longer than necessary, and must be deleted or anonymized when no longer required for the purposes for which they are processed (Art. 5(1)(e)).

To be compliant with the GDPR, data controllers – i.e., the legal or natural persons who determine for which purposes and with which means personal data are processed – must comply with these and other principles and be able to demonstrate compliance with them (Art. 5(2)). In addition, data subjects have extensive rights regarding the data concerning them, including the right of access (Art. 15), rectification (Art. 16), erasure (Art. 17), restriction of processing (Art. 18), data portability (Art. 20) and the right to object (Art. 21). Data controllers must therefore be able to provide a data subject, upon request, with the meta-information defined in Art. 15(1) on the processing of the data concerning them, as well as a copy of the data themselves. The data controller, upon request of the data subject, must also be able to rectify the data if they are inaccurate; be able to erase the data if they may no longer be processed; and be able to restrict the processing for certain purposes. In addition, a data controller should be able to determine whether a particular piece of data needs to be transferred under the right to data portability.

Regulations similar to the GDPR exist around the world. To name a few: California's Consumer Privacy Act (CCPA), Japan's Act on the Protection of Personal Information (APPI) as well as South Africa's Protection of Personal Information Act (POPI Act) all contain similar principles, including provisions for purpose limitation.

2.2 Dynamic Information Flow Tracking

Information Flow tracking is concerned with measuring the propagation of data through a system. Studied since the 70s, e.g., [13], information flow control systems are used to ascertain both confidentiality and security properties of programs. They trace the flow of data from a *source*, e.g., user input, to functions with security or confidentiality implications, so-called *sinks*. This can either be done statically, by constructing and analysing the data flow graph of an application, or dynamically, by labeling data with metadata and propagating those labels during program execution. In this work, we concern ourselves mainly with dynamic taint tracking, a form of dynamic information flow tracking. In this approach, the tainting framework transparently attaches labels to program data when they leave a source function. Throughout the program's execution, the metadata structure must be correctly propagated, even during operations that copy or modify the original data. Upon reaching a sink, the metadata can be extracted and if the data flow into a certain sink is undesired, the runtime can act accordingly, e.g., by blocking the operation. The lifecycle and processing of the metadata can be divided into three steps: *Introduction*, referring to the selective tainting of data at sources using defined policies. *Propagation* which refers to the preservation of metadata when the data are processed. *Checking* denotes the evaluation of metadata entering sinks and acting upon unintended data flows [40]. A wide range of academic work has covered automated vulnerability [e.g., 20, 24, 32] and privacy leak [e.g., 12, 55] detection with tainting.

3 SOFTWARE REQUIREMENTS FOR GDPR

Our goal is to design a framework to aid GDPR compliance for web applications via non-intrusive dynamic data-flow tracking. In order to do this, we first we examine the obligations set out by the GDPR with relation to software.

3.1 Roles and Threat Model

We start by defining four key roles: data subjects (e.g., users), the data controller, data recipients and developers. Personal data concerning a data subject is collected and processed by the data controller via code written by the developers. The data can further be processed by the data recipients for certain purposes. The data subject expects the data controller to process personal data in a compliant way, which requires the data controller to comply with their obligations under the GDPR. On the other hand, both data recipients and developers are honest but reckless, meaning they might unintentionally analyze data (e.g., via software bugs) in a way which violates data privacy regulations or add features that are add odds with said regulations. Traditionally, fixing these bugs requires additional effort on behalf of the data controller (via software development) to ensure compliance. We aim to minimize this effort by automatically controlling the flow of personal data in the application. To do so we assume that the data controller has full transparency into the web application's functionality, but may not have the resources or ability to fix privacy violating code (e.g., due to closed third-party libraries). Note that it is not our aim to protect against malicious data controllers or recipients who intentionally introduce compliance violations into their code.

3.2 Compliance with Predefined Policies

The information requirements in Art. 13 and Art. 14 oblige the data controller to determine the processing rules in advance and to provide certain information to the data subject, including the purposes of the processing, the recipients of the data, and the storage period. This leads to the following requirements:

(Req. 1) **Purpose Limitation** Software must process personal data only for previously defined purposes (Art. 5(1)(b), and consequence of Art. 13(1)(c), Art. 14(1)(c)).

(Req. 2) **Recipients** Software must transfer personal data only to specified recipients or categories of recipients (consequence of Art. 13(1)(e), Art. 14(1)(e)).

(Req. 3) **Storage Period** Software must provide a feature to erase personal data after the previously established storage period has expired or, if the storage period cannot be established beforehand, after the storage criteria do no longer apply (Art. 5(1)(c), and consequence of Art. 13(2)(a), Art. 14(2)(a)).

Other information obligations from Art. 13 and Art. 14 are merely informative in nature, without having a significant influence on the data processing in software.

3.3 Ensuring an Adequate Level of Protection

The risks to the rights and freedoms of natural persons posed by the processing of personal data vary depending on the context and must be taken into account when determining an adequate level of protection (Art. 32). For example, the processing of special categories of personal data under Art. 9 might require a higher level of protection than the processing of personal data of other categories. Hence:

(Req. 4) **Protection Level** Software must ensure at least an adequate level of protection when processing personal data. Note that the meaning of *adequate* is context specific and must be determined on a case-by-case basis.

3.4 Reacting to Data Subject Requests

Chapter 3 of the GDPR provides comprehensive data subject rights. Software must support the data controller in responding to data subject requests.

(Req. 5) **Right of Access** Software must allow the search of its database in such a way that the personal data concerning a data subject can be extracted. The data controller must be able to provide the data subject with a copy of the personal data concerning them (Art. 15(3)).

(Req. 6) **Context** Software must provide information on the processing purposes, category, recipients and storage period of personal data (Art. 15(1)(a–d)). In addition, if the personal data were not collected from the data subject, any available information about the origin of the data must also be provided (Art. 15(1)(g)).

(Req. 7) **Rectification** Software must allow the correction of inaccurate personal data concerning a data subject (Art. 16).

(Req. 8) **Erasure** Software must allow personal data concerning a data subject to be erased (Art. 17).

(Req. 9) **Restriction** Software must allow the restriction of processing of personal data and the ability to remove this restriction at a later point in time (Art. 18).

(Req. 10) **Actual Recipients** Software must enable the data controller to know the actual recipients of personal data (so that she can inform them of rectification, erasure and restriction of processing requests under Art. 19(1) and the data subject under Art. 19(2)).

(Req. 11) **Data Portability** Software must allow the export, in a common machine-readable format, of those personal data concerning a data subject that are subject to the right to data portability.

That is, data provided by the data subject to the data controller and processed on the basis of the data subject's consent or for the performance of a contract which the data subject is party to (Art. 20).

(Req. 12) Objection Software must not process personal data for certain purposes if the data subject objects to the processing of such purposes and such objection is justified (Art. 21, e.g., in case of direct marketing purposes, para. 3).

4 DATA PROTECTION VIA TAINT TRACKING

In the following, we first propose a metadata structure necessary to fulfill all twelve requirements defined above. We go on to describe a mechanism to explicitly associate application data with its metadata and propagate this information at run time using dynamic information flow tracking.

4.1 Metadata Structure

Based on the twelve software requirements, we define the metadata M as the tuple $\langle Q, f, l, S, i, p, r\rangle$ associated to personal data d as follows:

Purpose, Recipient, Period (PRP): Given a set U of possible purposes and a set V of possible data recipients to whom d may be transfered, we define $Q = \{q_0 \dots q_n\}$, where each $q_i = \langle u_i, v_i\rangle$ is a purpose-recipient pair such that $u_i \in U$ and $v_i \in V$. For each entry q_i, we also define a time period for which processing is allowed, described by the mapping $f : Q \to E$, where E is a set of valid time periods. To this end we also define a function `isValid(e)` which checks whether a given time period is valid.

Required Protection Level (RPL): The required protection level for d is defined as $l \in L$, where L is an ordered list of distinct protection levels. Note that the exact definition of L must be specified by the data controller for each application.

Data Subject (DS): A set $S = \{s_0 \dots s_n\}$ of data subjects associated with d. The value of s can be a pseudonym or identifier (such as a user ID) which uniquely defines users of an application.

Datum ID (DID): A unique identifier i to refer to d. We also define a function `newID()` which generates new unique identifiers.

Qualification for the right to portability (QFP): A boolean indicator $p \in \{0, 1\}$ of whether d must be transferred under the right to data portability. Practically this implies that for all data provided directly by a single user, $p = 1$.

Processing restricted (PR): Indication $r \in \{0, 1\}$ of whether the processing of d is restricted.

Table 1 maps the metadata attributes to the data-protection software requirements introduced in Section 3. To fully implement a requirement, all attributes associated with that requirement are necessary. If some of the requirements are redundant, e.g., because the data controller can handle its legal obligations in another way, the metadata can be adapted accordingly.

4.2 Making Implicit Meta Information Explicit

The ability to infer whether an arbitrary piece of application data is also personal data is practically impossible, and only becomes apparent when examining the context in which the data are used. For example, the string "corona" in an application may refer to a brand of beer in an e-commerce framework, or a medical diagnosis

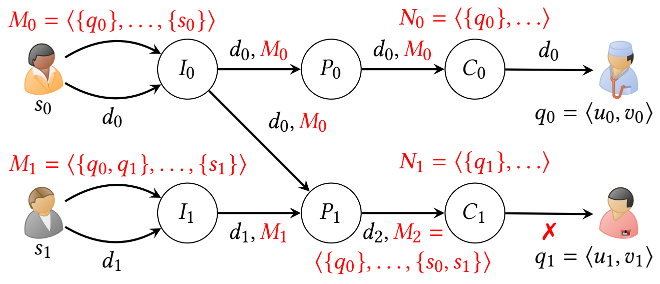
$M_0 = \langle\{q_0\}, \dots, \{s_0\}\rangle$
$N_0 = \langle\{q_0\}, \dots\rangle$
s_0
d_0
I_0
d_0, M_0
P_0
d_0, M_0
C_0
d_0
$q_0 = \langle u_0, v_0\rangle$
d_0, M_0
$M_1 = \langle\{q_0, q_1\}, \dots, \{s_1\}\rangle$
$N_1 = \langle\{q_1\}, \dots\rangle$
s_1
d_1
I_1
d_1, M_1
P_1
$d_2, M_2 =$
$\langle\{q_0\}, \dots, \{s_0, s_1\}\rangle$
C_1
✗
$q_1 = \langle u_1, v_1\rangle$

Figure 1: Introduction, propagation and checking of metadata during execution of a simple example application. Concepts introduced in this paper are shown in red.

for a patient in a medical record system. In other words, it is impossible to infer M by simple inspection of d: instead we require a mechanism to associate M with d throughout its lifetime in the application. However, modifying the application itself to provide this functionality directly may require major changes or may even be impossible in the case of third-party dependencies.

Therefore, we suggest a different approach: using non-intrusive dynamic information flow tracking to attach M to d at runtime. A high level example illustrating our approach is provided in Figure 1, which describes a simple hospital administration application. Here we introduce two data subjects Alice (s_0) and Bob (s_1), and two recipient-purpose pairs: a doctor using personal data for medical diagnosis (q_0) and a laboratory using personal data for a study (q_1). Both Alice and Bob submit metadata M_0 and M_1 respectively, summarizing their consent preferences.[1] While Bob allows his data to be used by both the doctor and the laboratory ($Q_1 = \{q_0, q_1\}$), Alice wishes to restrict her data usage to the doctor only ($Q_0 = \{q_0\}$).

First, personal data enters an application during the *introduction* step, where the information flow framework attaches metadata detailing the data subject's consent preferences. For example, in Figure 1, Alice (s_0) submits personal data d_0 to the application, together with metadata M_0 summarizing her consent preferences.

[1]Note that for illustrative purposes, only a subset of the metadata is shown in the diagram.

Table 1: Mapping Metadata to Software Requirements

Requirement	Purpose Limitation	Recipients	Storage Period	Protection Level	Right of Access	Context	Rectification	Erasure	Restriction	Actual Recipients	Data Portability	Objection
PRP	✓	✓	✓			✓						
RPL				✓								
DS					✓	✓	✓	✓	✓	✓	✓	
DID						✓				✓		
QFP											✓	
PR									✓			✓

In the introduction step (I_0), Alice's consent preferences (M_0) are attached to d_0 by the information flow tracking framework.

During program execution, the metadata are propagated alongside the actual data throughout the program's execution. For example, during the *propagation* step P_0 in Figure 1, the information flow tracking framework must ensure that the metadata M_0 remain associated to d_0 on its output edge. In reality P_0 could represent, for example, writing d_0 to a database, where the consent preferences must be correctly stored and subsequently re-attached when the data are retrieved. Special care has to be taken when personal data are combined, i.e., the application derives data from several pieces of personal data. We define the combination of two metadata $M_0 = \langle Q_0, f_0, l_0, S_0, i_0, p_0, r_0 \rangle$ and $M_1 = \langle Q_1, f_1, l_1, S_1, i_1, p_1, r_1 \rangle$ as $M_2 = M_0 + M_1$, with the propagation rules described in Table 2. Note that the data portability is always set to false when resolving conflicts as the result of the operation, the derived data, was not collected directly from the data subject, and therefore does not qualify for portability [2]. For example, in Figure 1, Bob (s_1) submits personal data d_1, which is introduced in I_1, and then combined with Alice's data (d_2) in propagation step P_1. In this case $M_2 = \langle \{q_0\} \cap \{q_0, q_1\} = \{q_0\}, \dots, \{s_0\} \cup \{s_1\} = \{s_0, s_1\} \rangle$, meaning d_2 is now associated to both Alice and Bob, with consent provided for q_0 only.

Finally, in the *checking* stage, the data controller must define a set of functions or interfaces in the application where data is processed for a specific purpose or disclosed to a particular recipient. Each of these functions is labeled with a tuple $N = \langle Q', l' \rangle$, where $Q' = \{q'_0 \dots q'_n\}$ is a set of purpose-recipient pairs: $q' = \langle u', v' \rangle$, where $u' \in U, v' \in V$ associated with the function, and l' is the level of protection provided by the function. When personal data with attached metadata M enters a function with intent label N, processing shall only be permitted under the following conditions: $Q' \subseteq Q$, in other words all processing purposes have been allowed by the metadata. In addition, we require that the periods for each purpose and recipient required by the function are all valid: defining $E' = \{e' | e' = f(q')\}$, we require that $\texttt{isValid}(e') = 1 \; \forall e' \in E'$. Finally, we require that the protection level provided by the function meets the protection level required by the metadata ($l' \geq l$) and that processing is not restricted ($r = 0$). Two examples are shown in Figure 1: during checking stage C_0, the data tracking framework successfully checks that $\{q_0\} \subseteq \{q_0\}$ and d_0 is transfered to the doctor. On the other hand, in C_1 the checking step fails as $\{q_0\} \nsubseteq \{q_1\}$ and d_3 is therefore not transferred to the laboratory.

Table 2: Propagation Rules for Conflicting Metadata.

Metadata	Resolution Rule
PRP	Purpose, Recipients: Intersection: $Q = Q_1 \cap Q_2$
	Period: Minimum: $f(q) = \min(f_1(q), f_2(q)), \forall q \in Q$
RPL	Maximum: $l = \max(l_1, l_2)$
DS	Union: $S = S_1 \cup S_2$
DID	Re-generate: $i = \text{newID}()$
QFP	Value always no: $p = 0$
PR	Or: $r = r_1 \vee r_2$

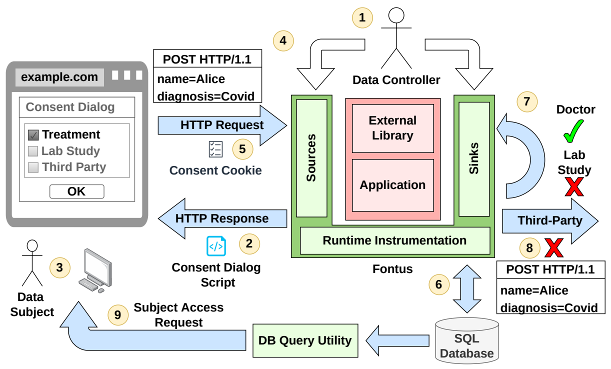

Figure 2: Data Protection via Dataflow Tracking

5 PROCESSING PERSONAL DATA

In this section we describe how the concepts defined in Section 4 can be used to construct a practical framework to aid GDPR compliance for web applications. A graphical overview of the approach is depicted in Figure 2, where we refer to steps ①–⑨ in the text below.

5.1 System Configuration

The framework must first be appropriately configured by the data controller before it can be deployed ①, as follows: Firstly, any functions which act as inputs for personal data (i.e., *sources*) must be identified so that metadata objects M are correctly attached at runtime. For example, in a web application, HTTP parameters from an HTML form containing fields such as name and address should be considered as personal data. In addition, a required protection level l has to be defined and attached to data leaving the source. Secondly, the data controller must also construct N for functions or interfaces where data are used for a specific purpose and/or by specific recipients (i.e., *sinks*). For each sink, an appropriate mitigation must also be defined to execute if processing of data with metadata M is not allowed.

Note that the data controller must have defined the purposes and recipients anyhow prior to the processing in their data protection policies by law, whether our approach is deployed or not. We therefore do not consider this step to introduce significant additional overhead for the data controller.

5.2 Consent Assessment

A key question is how to assess which purposes and recipients should be allowed for a particular piece of personal data. In other words, how should the content of Q be determined? To answer this question we divide the options stated in Art. 6(1) into three groups: user consent, user interaction and external circumstances.

User Consent via Cookie Dialogue. Art. 6(1)(a) states that one possibility for lawful data processing is consent from the data subject, with primary conditions for consent regulated in Art. 7. We implement this scenario by intercepting all outgoing HTTP responses and injecting a consent dialog script ②. On first visiting a website, the script triggers a pop-up asking the data subject to provide their consent preferences Q for the application ③. For example, Figure 2 shows the data subject (Alice) has only given consent for her personal data to be used to treat her symptoms. These preferences are encoded and stored by the script in a browser cookie which will be automatically sent to the application with every HTTP request ④. The tainting framework detects incoming personal data by instrumenting the source functions (typically HTTP requests) defined by the data controller. If the instrumented code detects an HTTP request containing personal data ⑤, the tainting framework can read the cookie, construct M and attach it to d. In case the user interacting with the website is not the actual data subject, the operator of the site needs to take care to ensure that the actual data subject's consent is available and correctly link it to the newly introduced data.

User Interaction. Through interaction with the application, the set of allowed purposes can change, e.g., if data processing is necessary to fulfill a contract with the user. Thus, in addition to the cookie based solution, our framework also allows one to dynamically augment allowed purposes resulting from particular interactions with the application and are not explicitly requested via a dialog. For example, consider an online shop where a shipping address is required to order items. During the order process the address can then be dynamically tagged with the purpose *shopping*. Thus, it is clear data was put into the system for the purpose of the order and may not be used, for example, for unrelated brochures with promotional offers.

External Circumstances. In the case of external circumstances, the processing of personal data is lawful if the data controller has legal obligations to comply with the operation in question, such as the release of connection data at the request of an authority. These legal obligations cannot be checked or applied to specific data by automated means and are therefore, considered as out of scope.

5.3 Persistence of Metadata

If personal data is persisted, for example via storage in a database, the associated metadata must also be stored ⑥. Then, when personal data is retrieved from the database at a later point in time (or by another application), the corresponding metadata must be restored and re-associated. This is especially important when using information flow tracking techniques to ascertain lawful processing, as the timespan between data introduction and the processing might span several years.

5.4 Preventing Unlawful Processing

In order to prevent processing of personal data for purposes which are not allowed, our framework inserts hook functions which are triggered before a sink function is executed. This hook function will first check if incoming data has attached metadata, and if so, whether it passes the checking requirements described in Section 4.2. Note that sinks checks can be performed for both internal processing of data ⑦ or when data is sent externally ⑧. If the checks are unsuccessful, the mitigation action associated with the sink is executed to ensure that data protection policies are not violated. Our framework allows the definition of arbitrary mitigation functions, and we describe two examples below:

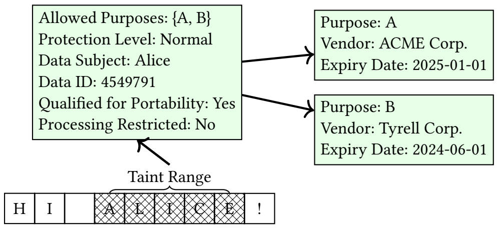

Figure 3: GDPR Taint Metadata

Removing Personal Data. The personal data which may not be processed at the sink is simply removed. For example, instead of the name "Alice", the framework would return an empty string or another value unrelated to any specific person.

Block Processing. Another option is to simply block execution of the processing operation. This can be achieved by throwing an exception or returning an empty value (i.e., `null` in Java) depending on the application's logic.

5.5 Rights of the Data Subject

Persisting the proposed metadata also allows the automatic realization of several additional key aspects of data protection regulations such as the GDPR. For example, access requests under Art. 15 can be fulfilled by searching for all database entries with associated metadata belonging to the requesting person ⑨. Similarly, the right to erasure under Art. 17 can be carried out by deleting all data associated with a given data subject. Additionally, the portability (Art. 20) of personal data can be ascertained by checking metadata QFP. Finally, the withdrawal of consent can be realized by modifying the purpose and recipient entries in metadata PRP for the requesting user.

6 IMPLEMENTATION

In the following, we describe our implementation of the approach introduced in Section 5, including details of the data-flow tracking engine, its persistence component, and how the resulting framework can be transparently applied to existing Java applications.

6.1 Taint Engine

To realize dynamic data-flow tracking in Java applications, we present Fontus, a non-intrusive taint tracking engine. Fontus implements tainting by rewriting the bytecode of Java classes when they are loaded by the JVM. This rewriting step replaces instances of string-like data types with our own *taint-aware* versions. This encompasses both data types directly storing strings, such as `String`, `StringBuilder` but also data types manipulating strings such as `Formatter`, `Pattern` and `Matcher`. The taint-aware classes contain two fields: a reference to an underlying string-like instance, and an associated taint metadata object. Bytecode rewriting ensures that our technique is compatible with arbitrary JVM implementations, avoiding costs of maintaining a modified JVM, and allowing deployment in scenarios where JVM choice is restricted, such as platform-as-a-service offerings.

As shown in Figure 3, the taint metadata is structured as a list of *taint ranges*, refering to a (sub) sequence of the string's characters. Each range itself contains an implementation of the metadata

```java
public final class TaintString {
  private String string;
  private TaintMetadata metadata;

  public TaintString concat(TaintString tstr) {
    // Propagate metadata
    TaintMetadata meta = metadata.concat(tstr.metadata);
    // Perform string operation
    return new TaintString(this.string.concat(tstr.string), meta);
  }
}
```

Figure 4: Illustrative method from a taint aware string class

described in Section 4. This high level of granularity allows to evaluate the privacy implications of operations on a per-character basis, even if data from several data subjects are combined into one string. Each taint-aware method call performs two operations: calling the equivalent method on the wrapped string-like object, and ensuring metadata is propagated appropriately. For example, Figure 4 shows a simplified example of the `concat` method. By essentially proxying method calls, we can enable taint tracking while benefiting from extensive performance optimizations of the JVM. For example, the HotSpot JVM will replace commonly executed string concatenations with a native implementation. We also support taint propagation for reflected method calls by proxying e.g., the `invoke` method and performing conversion to taint-aware types at runtime. In addition to class replacement, Fontus also implements application specific sources and sinks for the introduction and checking of metadata via insertion of hook functions at bytecode level.

One challenge of this approach is that the Java standard library classes are loaded before Fontus itself, and therefore cannot be modified to accept our taint aware strings. Instead, we unwrap all taint aware classes when passing them to standard library functions. In order to prevent loss of taint metadata during calls to standard library methods, we proxy these calls to ensure taint information is correctly propagated. For example, the `StringJoiner` class allows easy creation of a string containing a list of values separated by a delimiter. To correctly propagate the taint, we redirect these calls to a Fontus runtime class which computes the taint and calls `StringJoiner` internally.

6.2 Taint Persistence

Most tainting engines only track taints of values residing in the application's heap memory. While this approach may be sufficient for security purposes, it is insufficient for data protection enforcement. Take for example an online shop which stores a customer's e-mail address when an order is placed. If the company decides at some point in the future to use stored e-mail addresses to send advertisements, this can constitute a GDPR violation. If taint information were removed upon entering the database, detecting such a violation would be impossible. Therefore, our taint engine persists taint information in the database. This allows us to enforce privacy properties across multiple sessions or even between multiple applications sharing the same data source.

Enabling taint persistence has two requirements: preprocessing, which is performed offline prior to starting the application, and query rewriting at runtime.

6.2.1 *Preprocessing.*

The preprocessing step modifies the database schema to make it taint aware. Here we modify the structure of every database table to allow storing the taints next to the actual values. For each column in a database table, the preprocessor adds a second column to store taint information. These newly introduced taint columns are denoted with the `t` subscript. As this process simply adds additional columns, this representation is fully database agnostic.

6.2.2 *Query Rewriting.*

In addition to creating the taint columns, all queries interacting with the database have to be made taint aware as well. Fontus thus includes a Java Database Connectivity (JDBC) driver which acts as a wrapper for the JDBC driver actually used by the application. Our driver dynamically rewrites all incoming SQL statements and passes them on to the wrapped driver.

The query rewriting works as follows: The columns affected by an SQL query are separated into three different categories. The result set, that is the columns returned by e.g., a `SELECT` statement. The update set, which are the values written by `UPDATE` or `INSERT` statements and the condition set, that is the values occurring inside e.g., `WHERE` clauses. For example:

$$\texttt{SELECT } \overbrace{\boxed{\texttt{a, b, d}}}^{\text{result set}} \texttt{ from t WHERE } \overbrace{\boxed{\texttt{id = ?}}}^{\text{condition set}}\texttt{;}$$

$$\texttt{UPDATE } \underbrace{\boxed{\texttt{a = ?}}}_{\text{update set}} \texttt{ in t WHERE } \underbrace{\boxed{\texttt{id = ?}}}_{\text{condition set}}\texttt{;}$$

For all elements in both the result as well as the update set their corresponding taint values need to be considered as well. The taint value of the elements in the condition set are not considered. Thus both queries are transformed as follows:

$$\texttt{SELECT } \overbrace{\boxed{\texttt{a, } a_t\texttt{, b, } b_t\texttt{, d, } d_t}}^{\text{result set}} \texttt{ from t WHERE } \overbrace{\boxed{\texttt{id = ?}}}^{\text{condition set}}\texttt{;}$$

$$\texttt{UPDATE } \underbrace{\boxed{\texttt{a = ?, } a_t \texttt{ = ?}}}_{\text{update set}} \texttt{ in t WHERE } \underbrace{\boxed{\texttt{id = ?}}}_{\text{condition set}}\texttt{;}$$

Special consideration has to be taken when nested queries are involved. For example, consider the following query:

```sql
SELECT d, (SELECT COUNT(id) FROM t) AS e FROM t;
```

Here the simple transformation shown before does not work, as a nested `SELECT` query can only return a single column. Thus, the nested query has to be duplicated as a whole. This results in the following query:

```sql
SELECT d, d_t, (SELECT COUNT(id) FROM t) AS e, (SELECT '0' from t
↪  LIMIT 1) as e_t FROM t;
```

Untainted values are denoted as `'0'` while tainted values contain the serialized taint object as a JSON string.

The taint persistence has to be fully transparent for the application. For example, index based access to parameters and resulting values is common in the JDBC API. Thus, our taint driver has to adjust the indices of both query parameters and result set return values to keep the developer's intention intact. Additionally, all operations requesting information about the database schema have to return a view without the taint columns to not break code that has assumptions about e.g., a table's column order.

When setting a parameter's value the driver transparently sets the taint column to the serialized taint information and restores it upon read access.

```
1 long signUp(String name, String email) {
2   name = THandler.taint(this, name); // Fontus
3   email = THandler.taint(this, email); // Fontus
4   return UserDao.storeUser(name, email);
5 }
```

(a) Metadata Introduction

```
1 for(User user : getUsers()) {
2   String name = user.getName();
3   String header = String.format("Hi %s!", name);
4   String mail = user.getEmail();
5   THandler.check(NEWSL, header, mail); // Fontus
6   this.sendNewsletter(header, mail);
7 }
```

(b) Policy Enforcement

Figure 5: Enforcing Purpose Limitation

Table 3: Evaluation Applications

		LoC[†]			Handler Size	
Name	Ver.	Total	Java	Classes[‡]	LoC[†]	Effort
OpenOlat	16.1a	1.4M	1.1M	22k	125	-
Broadleaf	6.1.7	353k	201k	18k	189	-
OpenMRS	2.13	236k	127k	40k	285	-
OpenHospital	1.12.0	255k	59k	22k	167	5 h
JForum2	2.8.2	67k	35k	10k	71	3 h
CAP Bookstore	1.0.0	147k	12k	16k	82	7 h
HMSA-CTT	0.0.1	9k	4k	20k	129	6 h

(†) Lines of code measured by cloc (https://github.com/AlDanial/cloc).
(‡) Total number of loaded classes after running our tests.

This approach is both database agnostic as well as agnostic to frameworks for database interaction. For example, all applications used to showcase the feasibility of our framework use object relational managers such as Hibernate. As long as they use a JDBC driver for the underlying communication with the database, our framework can make the database interaction taint aware.

6.3 GDPR Tainting for Real Applications

Enabling taint tracking with Fontus for any Java application requires addition of the `javaagent` flag to its invocation, loading the taint engine itself, and adjusting the database setup to enable taint persistence. The latter requires preprocessing of the database schema and modification of all JDBC connection strings to use our driver. To enforce data protection properties three additional configuration steps are required:

Determine sources. Identify the operations where data enters the application. Typical instances of sources are reading HTTP request parameters or its body values. Additionally, operations where processing takes place that alters the privacy metadata shall be declared as sources too.

Define sinks. Identify operations where data leaves the application or processing with a specific purpose takes place. These sinks require a list of required purposes, the corresponding vendors performing the data processing, the allowed protection level of incoming data as well as a mitigation policy in case of a violation.

Taint Handler. While defining sources and sinks is sufficient for ensuring security properties, the taint metadata shown in Figure 3 requires dynamic information, e.g., to determine the actual data subject. Usually this is the user currently logged into the application, but there are more complex scenarios, too. For example, at the registration desk of a hospital a clerk enters data on behalf of the user. Thus, the data subject is not the logged-in user, i.e., the clerk, but the person seeking medical help. Retrieving these information is highly application specific and thus requires custom logic.

Therefore, sources are annotated with a so called *Taint Handler* responsible for retrieving this information and computing the correct GDPR taint value. When tainted data reaches a sink, Fontus executes its default GDPR policy which first removes all data where the processing restricted flag is set by replacing it with placeholder values and then validates that the purpose and vendor pairs required for the sink are allowed based on the taint data. If a violation is detected, the default policy triggers the mitigation policies of the sink one after another. In case the default GDPR policy is insufficient to handle the violation, e.g., due to the application expecting a specific exception, it is also possible to insert a call back to the handler here as well. This allows implementing application or sink specific mitigation approaches. When detecting a source or sink invocation in the bytecode, Fontus automatically injects calls to the Taint Handler and GDPR policy functions based on the configuration.

6.4 Full Example

A complete example on how Fontus enforces purpose limitation is provided in Figure 5. The lines with a comment at the end are those transparently inserted into the bytecode by Fontus. It first shows the sign-up process of a web application in Figure 5a. The `signUp` method is declared as a source, thus our framework inserts the TaintHandler calls on line 2 and 3. Here the framework retrieves the taint metadata from the configuration, environment and the current application state. On line 4 the newly entered data is stored in the database along with its taint metadata. The functionality shown in Figure 5b represents sending a newsletter to all users. On line 1 all users are retrieved from the database. As they have attributes constituting personal data, the corresponding taint metadata is restored as well. Then, on line 3 the tainted value `name` is combined with static data to form a string tainted such as shown earlier in Figure 3. In line 6 the application attempts to send the newsletter to the currently selected user. The `sendNewsletter` function is declared as a sink, thus Fontus inserts a call to the TaintHandler on line 5 to check whether the operation is permitted. Here the intent label N of the sink is checked against the metadata attached to this user's data. The sink's label is retrieved via its ID (here `NEWSL`) based on the provided configuration. If this check determines that processing is not permitted, the runtime can prevent the operation and consequently prevent a data protection infringement.

7 PRACTICAL EVALUATION

To demonstrate the feasibility of our approach, we configured, instrumented and deployed 7 open source Java applications using our compliance framework. We chose large open source applications with real world personal data usage to evaluate the applicability of Fontus to realistic data protection scenarios.

7.1 Selected Applications

The 7 selected applications are listed in Table 3 and described in more detail in the following.

OpenOlat [16] is an e-learning platform, used in both educational and commercial institutions. It can manage courses, exams, how users progress throughout their curriculum and offers collaborative features such as instant messaging and message boards. Additionally, it provides interfaces to a wide array of external tools such as Microsoft Teams or PayPal. It handles data with a broad spectrum of sensitivities, from less sensitive, e.g., instant messages between colleagues in a course, to highly sensitive, e.g., grades or information about disabilities.

Broadleaf [5] is an e-commerce solution, used for example by the Major League Baseball (MLB) organization to handle millions of subscriptions for the MLB's online services [6]. A noteworthy aspect of Broadleaf is its microservice based architecture. That is, it consists of several independent services sharing a central database. The services deployed throughout our testing are the customer facing website, an administrative website used to handle e.g., customer service requests and a service providing an application programming interface (API) for usage by external partners. Propagating taint information from e.g., the website upon purchasing an item toward the API requires our taint engine to persist the taint information in the central database.

OpenMRS [8] is a medical record system designed to handle the information technology requirements of hospitals. It is deployed in over 40 countries and handles the data of almost 15 million patients. It is designed to process patient information, including sensitive data such as medical condidtions and allergies. In contrast to the other use-cases, the actual data subjects (i.e., the patients), do not interact with the application themselves. Instead, only employees of the hospital and its partners directly interact with the web application. Our framework thus has to be able to handle such delegated forms of data entry.

OpenHospital [21] is a second medical record system. Unlike OpenMRS, its frontend is not build on Java technologies. Instead, it provides a REST API with a JavaScript web UI interacting with the backend.

JForum2 [22] is a web forum application, used to build and maintain a community. It stores personal information such as IP and e-mail addresses for moderation and notification purposes.

CAP Bookstore [39] demonstrates how to build business applications using the Cloud Application Programming Model framework with a book shop web application as an example. Users can purchase books and leave reviews, managed by a separate admin interface. Similar to OpenHospital, it provides its functionality via a REST API and a JavaScript frontend. Unlike the other applications, the CAP Bookstore uses an in-memory H2 database, which does not allow external database queries.

HMSA-CTT [48] Is an open source contact tracing solution developed by a German university of applied sciences. When attending a lecture each student has to check in on the HMSA-CTT website for the current room. In case of an infection, administrative staff can then generate a report of all users which were checked into the same rooms as the infected user.

Table 4: Mapping Software Requirements to Use Cases

Use Case	Purpose Limitation	Recipients	Storage Period	Protection Level	Right of Access	Context	Rectification	Erasure	Restriction	Actual Recipients	Data Portability	Objection
UC1			✓	✓				✓				
UC2									✓			✓
UC3	✓	✓										
UC4										✓		
UC5	✓			✓								
UC6	✓	✓		✓								
UC7	✓	✓										
UC8	✓	✓										
UC9	✓	✓										
UC10				✓								
UC11					✓	✓	✓	✓			✓	
UC12	✓	✓	✓		✓					✓		

7.2 Use Cases

For each application, we constructed one or two representative use cases to evaluate our methodology, with a further two generic use cases which are applicable to all applications. As shown in Table 4, the use cases have been chosen to allow evaluation of all GDPR requirements presented in Section 3.

UC1: OpenOlat – Contact Tracing
To combat the COVID-19 pandemic, OpenOlat includes contact tracing functionality, allowing users to check in when physically attending courses. In cases of a COVID infection, an administrator can generate reports to warn other attendees. These data are sensitive, as they can be used to track the movement of users and infer personal details about them. Thus, they should not be used for any purposes other than "contact tracing". Additionally, they should be deleted as soon as they are no longer required. These requirements should be automatically enforced by our approach. Note that this use-case equally applies to the HMSA-CTT application.

UC2: OpenOlat – Contesting Incorrect Data
In the default configuration OpenOlat does not allow users to change either first or last name. If the stored name becomes inaccurate, for example due to marriage, the user has to contact the system operator to rectify this mismatch. Under Art. 16, a data subject has the right to demand rectification of inaccurate data. For example, a wrong surname could be contested by the data subject and must not be used for further processing. Our framework shall ensure that contested data is not used for further processing until it has been corrected.

UC3: Broadleaf – Restricting 3rd Party Data Processing
Broadleaf offers a REST API connected to the store's database, allowing external tools to interact with the data in an automated fashion. An example for such an endpoint is the option to query customer profiles. In this way, it is possible to provide access to third-party providers who want to process the customer data, for example, for marketing purposes. This data includes, among others, the full name and saved addresses. To process customer data in a GDPR-compliant manner, a third party requires a legal bases, such as the customer's consent. Thus, a user has to consent to the "data

processing" purpose for the vendor "ACME" such that their data can be accessed and processed by said third party.

UC4: Broadleaf – Document 3rd Party Data Processing
According to Art. 19, the data controller shall communicate the rectification, erasure or the restriction of processing of personal data to any recipient to whom this information has been disclosed. Thus, for all personal data collected in UC3, it should be recorded when and to which 3rd party they were sent in order to be able to forward potential corrections or restrictions at a later stage.

UC5: OpenMRS – User Access to Sensitive Medical Data
OpenMRS stores and processes sensitive medical information of patients such as medical diagnoses and conditions. These data should only be visible to OpenMRS users with a need to view them, such as doctors and nurses, and remain hidden to other users, such as IT administration staff. Therefore a patient must consent to a "data processing" purpose for a number of recipients "doctor", "nurse", "clerk" etc. for their data to be visible.

UC6: OpenMRS – External Data Processing
OpenMRS allows patient data to be exported to other locations and external systems via the Fast Healthcare Interoperability Resources (FHIR) interface [15]. FHIR uses a RESTful protocol to allow interoperability between health care systems and provide information across a wide variety of devices. As FHIR shares data with third-parties, patients should explicitly give their consent for an "export" purpose before their data are transmitted via FHIR.

UC7: OpenHospital – User Access to Sensitive Medical Data
Similar to UC5, the OpenHospital application processes sensitive medical information, these data should only be visible to appropriate users.

UC8: JForum2 – Restrict Automated E-mail Notifications
JForum2 notifies users about replies to their posts via e-mail by default. While users have the possibility to disable these notifications via their profile, forum administrators can override this preference freely. By defining a purpose "e-mail notifications", users shall have full control over whether they receive notifications, regardless of administrative staff editing their preferences.

UC9: CAP Bookstore – Avoid Exposing Contact Information
The CAP Bookstore application allows users to leave book reviews and ratings. These ratings are public by default, and expose sensitive user contact information such as e-mail addresses. Users must therefore explicitly consent to a "review" purpose before their contact information can be viewed by other users.

UC10: HMSA-CTT – Encryption of Contact Information
As HMSA-CTT handles contact information related to medical infections, it could be considered sensitive data under GDPR. Any contact information should require a "contact tracing" purpose and stored in an encrypted form to meet Req. 4, PROTECTION LEVEL.

UC11: Generic – Subject Access Request
As postulated in Art. 15, data subjects have the right to request information about whether a data controller processes personal information concerning her and if so gain access to it. This is called a *subject access request* (SAR).

UC12: Generic – Records of Processing Activities According to Art. 30, each data controller has to maintain a record on the specifics of the application's data processing. This record has to include, for example, a list of other parties data is shared with, categories of stored data, or the expected timeframe for data deletion.

7.3 Evaluation: Enforcement

In this section we describe how Fontus was used to successfully implement the use cases described above. For each application we implemented the required taint handler code to correctly initialize the metadata according to the use cases and configured corresponding sources and sinks. The total amount of custom code in Fontus to support these use cases is listed in Table 3.

In addition, each user-facing application was placed behind an nginx reverse proxy which automatically injects the consent dialog and cookie as described in Section 5.2. Note that we did not change the source code of the stated applications.

OpenOlat. To realize UC1 and UC2, we labeled all data retrieved from HTTP requests as sources. Special care was needed for form data from POST requests, which OpenOlat reads as Java streams. In this case we marked the stream to string conversion functions as sources. For sinks, we identified the getter function of user classes as well as the getters of classes containing the user's attributes.

Fontus was correctly able to mark the contact tracing information with a high protection level and an expiry date of four weeks into the future. By querying metadata in the OpenOlat database with a custom utility tool (called *dbquery* and part of Fontus), we were able to successfully extract and delete contact tracing data for which processing was no longer allowed. This automated deletion enforces compliance with Req. 3. We were also able to implement and successfully test this use-case for the HMSA-CTT application.

Additionally, dbquery allows setting the restricted flag (r) for specific database entries in order to fulfill UC2. In this scenario, a user contacts the data controller to inform them of incorrect data (e.g., a name change). The data controller then uses dbquery to set the restricted flag $r = 1$ for the incorrect database entries associated with that user. We can therefore successfully prevent processing of contested data as Fontus checks the restricted flag at every sink. This enables automated fulfillment of Req. 9 and Req. 12. If the check fails we replace the contested value with a placeholder to avoid interfering with the execution of the application.

Broadleaf. The REST API of Broadleaf has no inherent restrictions on what data can be queried, such that a third party with access to the API has visibility over the entire customer database. In other words, all customers would have to give consent to data sharing in order for the REST API to be used in a compliant fashion.

To demonstrate Fontus can restrict API access to customers who have not given their consent (UC3), we first identified HTTP request parameters as source functions, with API calls to the database using so-called *Database Access Objects* as sinks. If an API query tries to access data from a customer who had not given their consent, the taint handler replaces the original API response with `null` to mimic an empty result. Note that in this case simply throwing an exception would result in an HTTP error which would still leak information (i.e., her existence) about the user. We also demonstrated that Fontus can be used to minimize sharing of unnecessary data by only allowing a subset of customer data to be shared: for example, if a third party performs marketing solely via e-mail, we can block access to the user's postal address. We therefore successfully demonstrate Req. 1 and Req. 2.

Additionally, we were able to demonstrate UC4 by adding the methods of the API itself as sinks. In this case we implemented a taint handler which logs information on the sink configuration and the unique ID (*i*) of the data. In this way Fontus keeps track of all personal data (including the associated data subject) shared with third parties. This allows the data controller to easily identify the third parties data has been shared with (fulfilling Req. 10) and forward all rectifications, erasures or restrictions of processing.

OpenMRS. OpenMRS already has a detailed user management and access control framework in place to restrict the visibility of sensitive data. We confirmed this by checking that a user with the *System Administrator* role was not allowed to view patient information via the standard web application. However, we discovered a legacy user-interface in OpenMRS where access policies are not correctly enforced and allow unprivileged users to view patient information including their medical conditions. In addition, we confirmed that third parties with access to the FHIR API also have full access to all patient data.

To realize UC5 and UC6, we identified two groups of sources, namely functions retrieving parameters from HTTP requests and HTML form entries. Similarly to OpenOlat, the form entries are read through a wrapper class, and as such we selected methods extracting string values from the wrapper as sources. As sinks, we identified the getter of the patient, diagnosis and allergen model classes as these are used for both display and search functionality (UC5). To provide restrictions on whose data is included in FHIR responses, we marked the functions setting properties of the report objects as sinks too.

We showed that Fontus can protect sensitive patient data in spite of application bugs or misconfiguration as follows. When registering a new patient, a dialog is presented asking the patient to provide their consent for sharing their data with the users of the FHIR API as well as the purpose *data-processing*, with recipients corresponding to roles (e.g., clerk, doctor, IT staff) of the OpenMRS application. When the data are subsequently retrieved at a later point, Fontus checks whether the patient has given their consent for the currently logged-in user to process their data. If consent has not been given, the patient information is automatically redacted and replaced by a placeholder. We therefore confirmed that using Fontus indeed prevents leaking patient information even in the presence of the bug mentioned above (UC5). Additionally, we could show that Fontus can successfully block calls to the FHIR API for patients who have not given their consent (UC6). We therefore successfully demonstrate Req. 1, Req. 2 and Req. 4.

OpenHospital. Similar to OpenMRS, any data related to the patient (via POST to the `patients` API) was tainted with the patient's consent preferences of who was allowed to access their data. If no consent had been given for a user to view patient data, we confirmed that patient information was no longer visible to that user (UC7), fulfilling Req. 1 and Req. 2.

JForum2. To realize the JForum2 specific use case, we identified data read from HTTP requests as sources. This encompasses query parameters, the user's IP address as well as the request body. UC8 is concerned with sending e-mails, thus it was necessary to identify both where e-mails are sent and how a user's e-mail address is read. While the e-mail address is used for several reasons, such as determining whether a user is banned. The combination of both functions allows to prevent sending notifications (fullfilling Req. 1) without violating invariants and break the application.

CAP Bookstore. To prevent exposure of a user's contact information without their constent, we instrumented all methods retrieving e-mail addresses as sources. Sink functions were identified as JSON serialization functions for REST API responses. We were able to show that Req. 1 and Req. 2 were fulfilled by confirming that e-mail addresses were only made public if the user had given their explicit consent (UC9).

HMSA-CTT. We identified all HTTP request parameters related to personal data (e.g., name, e-mail, address) as sources and marked them as sensitive with an expiry date of two weeks. To realize UC10, we ensured that all sensitive data written to the database was encrypted (Req. 4), and subsequently decrypted on retrieval. Note that although HMSA-CTT can encrypt contact information natively, we are able to guarantee this protection even if the feature is disabled.

Generic use cases spanning all three applications. Finally, we report on our experiments concerning the generic use-cases, where we rely on general, application-agnostic functionality provided by Fontus. These final use cases demonstrate fulfillment of a number of requirements as shown in Table 4.

To successfully comply with a subject access request (UC11), all personal data concerning a specific user has to be retrieved. In non-trivial applications, this data is distributed across a multitude of database tables and columns. Thus, extracting the data to conduct such a subject access request is an elaborate task if no automation is in place. Fontus can assist in performing subject access requests as all personal data stored in the database will be annotated with meta-data which contains information about the associated data subject. Thus, data belonging to a user can be trivially queried and made available to the requesting person. For all selected applications bar CAP Bookstore, we were able to create subject access request procedures with the aforementioned dbquery utility, which successfully retrieved all stored information for a given user. Creating these did not require any knowledge about the actual application logic or the intricacies of the database layout.

In respect to the required record of data processing activities (UC12), key parts (Art. 30(1)(b), (c) and (d)) can be retrieved directly from the Fontus configuration. For instance, while Fontus is active, it is impossible to share data with external parties that are not explicitly enabled in the configuration. This has two distinctive advantages: For one, the creation of the record can be partly automated. Furthermore, it is guaranteed that the record represents the actual behavior of the application. In this way the provided record and the application's actual processing practices are guaranteed to remain synchronized and correct.

7.4 Evaluation: Robustness

We also performed a number of measurements designed to evaluate the robustness and completeness of our approach. For each application we wrote end-to-end tests designed to maximize exploration of the target application, whilst incorporating the use-cases described

Table 5: Robustness test summary

Application	Tainting Coverage			Code Coverage†			DB Size (kB)‡		SQL Queries		SQL Tokens	
Name	Source	Sink	Tainted	Class	Method	Line	Startup	Test	Total	Unique	Original	Rewritten
OpenOlat	75.0%	75.0%	23.3%	63.6%	37.4%	31.4%	11728	12224	8713	654	178.60	321.95
Broadleaf	100.0%	100.0%	40.0%	60.5%	31.6%	28.4%	8048	115408	193362	442	71.77	125.80
OpenMRS	100.0%	86.0%	38.2%	68.9%	33.2%	27.6%	18032	18224	119539	506	173.06	316.01
OpenHospital	100.0%	100.0%	1.0%	78.7%	55.7%	42.5%	1785	2153	450	54	126.19	210.48
JForum2	100.0%	100.0%	47.5%	78.5%	75.2%	66.1%	1472	1984	6281	239	24.83	37.98
CAP Bookstore	100.0%	100.0%	2.0%	100.0%	76.9%	73.9%	-	-	1329	579	60.26	92.10
HMSA-CTT	75.0%	100.0%	13.0%	71.4%	54.1%	60.9%	192	224	116	18	41.28	63.83

†: Measured using Jacoco (https://www.eclemma.org/jacoco/), ‡: Computed by summing data and index size for all tables in the database.

above, resulting in a total of 229 test cases. These tests were performed from the perspective of a web user and executed using the Playwright browser automation framework [33]. For each application we ran the corresponding tests and measured the metrics described below and summarized in Table 5.

7.4.1 Test Coverage. To ensure that our tests are an effective way of evaluating the use cases in Section 7.2, we report the percentage of source and sink functions triggered at least once during test execution (*Source* and *Sink* columns in Table 5). For all applications, almost all sources are triggered, and either all or a high percentage of sinks are executed. This shows that Fontus is successfully introducing and checking for tainted data at application runtime. We also report the percentage of sink executions with tainted data entering the sink (*Tainted* column in Table 5). For example, OpenHospital sinks check whether data entering the HTTP response body are tainted. As the response is in JSON format, we would expect a mix of untainted (e.g., for field names) and tainted (for values containing personal data) events. We indeed confirm that both tainted and untainted data are detected by Fontus for each application. The large variation in the tainted fraction is due to the nature of the application and how efficient the test cases are in triggering sinks with tainted inputs. We also report the code coverage of classes in the application's name space recorded after testing to highlight that our tests sufficiently explore the target applications. On average our tests executed code in 75% of application classes.

7.4.2 SQL Rewriting. In order to evaluate the effectiveness of SQL rewriting we measured the total amount of data written to the SQL database after application startup and after test execution. For the CAP Bookstore application, database size measurements were not possible due to the deployment of an embedded in-memory H2 database. In all cases we observe an increase in size, confirming the Fontus successfully writes database entries. The numbers also confirm the scalability of our rewriting engine, which successfully processed queries for databases ranging in size from a few hundred kB up to 107 Mb. In one case (Broadleaf), the engine was able to process queries for over 100 Mb of data during test execution.

In addition, we measured the total number of SQL queries executed by each application, together with the number of unique queries rewritten by our engine. The results in Table 5 show that our SQL rewriting technique is not only able to successfully handle a large volume of queries, but also a large variety of different queries. The complexity of SQL queries was further assessed by counting the number of SQL tokens present in each query. For example, the simple query `SELECT * from table;` has a token count of 5. For each application we measured the average number of tokens before and after our rewriting. The results show that our engine is capable of handling complex queries with many hundreds of SQL tokens.

7.5 Evaluation: Performance

Table 6 summarizes the performance overhead of Fontus. All applications ran on OpenJDK 11's HotSpot JVM and, together with their respective databases, were hosted in Docker containers on a machine with an AMD EPYC 7702P 64 core CPU and 512GB RAM.

For the three largest applications we picked a sequence of typical interactions (described in the supplemental) and executed them, for multiple concurrent users, in a loop, measuring the average HTTP response time. All testing was done headless with the Gatling performance testing framework [17], meaning that no client side rendering is included in the numbers. Each measurement was divided into two phases. First, the warmup phase where the sequence is ran for one minute with one user to ensure the JVM has loaded all required classes and optimized their code. Then, in the measurement phase several concurrent users perform their actions in a loop for three minutes. The reported numbers are from the latter phase, emulating the typical state of a web service taking user requests.

For OpenMRS, the application itself (without Fontus instrumentation) became unstable and was prone to crashes for more than 50 users. Therefore, we have only reported numbers for up to 50 concurrent users. We also computed the storage overhead (Column 'S' in Table 6) for the deployment of OpenMRS with Fontus.

Table 6: Runtime Overhead induced by Fontus

U	OpenOlat			Broadleaf			OpenMRS			
	R	**T**	**O**	**R**	**T**	**O**	**R**	**T**	**O**	**S**
1	10 ±12	16 ±15	6 60.0%	43 ±62	57 ±62	14 32.6%	121 ±157	175 ±220	54 44.6%	3.5%
10	10 ±13	16 ±16	6 60.0%	43 ±65	59 ±70	16 37.2%	117 ±168	160 ±223	43 36.8%	42.8%
25	11 ±16	18 ±20	7 63.6%	48 ±65	69 ±79	21 43.8%	121 ±184	152 ±219	31 25.6%	59.4%
50	16 ±23	22 ±24	6 37.5%	61 ±80	93 ±137	32 52.5%	119 ±185	144 ±200	25 21.0%	49.6%
75	21 ±27	27 ±32	6 28.6%	67 ±90	125 ±216	58 86.6%	-	-	-	-
100	26 ±32	34 ±33	8 30.8%	79 ±80	138 ±222	57 72.2%	-	-	-	-

Legend: **U**sers, **R**egular, **T**ainted, **O**verhead, **S**torage overhead.
All timing measurements are provided in milli seconds.

Our results show that Fontus has excellent potential as a practical solution for runtime data protection enforcement. In each application, the overhead caused by our framework is within the standard deviation of the untainted measurement. Although a direct comparison is not possible, (due to, e.g., differences in supported features, metadata structure and Java versions), our results are broadly comparable with previous works using bytecode rewriting to implement dynamic taint tracking for Java. For example, Martin et al. [28] measure an average overhead of 57.4% for five web applications, while Bell and Kaiser [3] obtain an average overhead of 53.3% across two Java benchmarking suites. A more detailed discussion of the overhead caused by different parts of Fontus can be found in the supplemental document. Although achieving overheads for deployment in a production environment was not the focus of this paper, we believe that further performance improvements are possible by implementing additional features such as optimized caching of SQL transformations or selective instrumentation via static analysis.

8 DISCUSSION

Our experiments have shown that Fontus can enhance an application to provide data protection guarantees without any changes to the application logic or source code. Thus, we have demonstrated that our solution can indeed aid data controllers in fulfilling their legal responsibilities under the GDPR, such as "to integrate the necessary safeguards into the processing in order to meet the requirements of this Regulation and protect the rights of data subjects" (Art. 25(1)). Additionally, the storage of the concerned data subject together with each piece of data allows a data controller to greatly simply mandated processes such as the record of processing activities or, in case of a data breach, to pinpoint affected data subjects.

To configure Fontus for the chosen applications, it was necessary to create a certain amount of application specific code, for example to realize the mitigation policies for the respective application. However, as shown in the "Handler" column of Table 3, the amount of custom code required is reasonably small, especially when compared to the effort which would be required to implement the same data protection features by hand. Additionally, the taint handler code already contains basic routines required to realize most use cases, such as determining the current user. Fontus loads the handler code directly by itself, so no modifications of any application's source code were necessary. This property of Fontus is significant, as it enables us to prevent data protection violations even in the presence of binary only dependencies. Identifying both sources and sinks was possible without prior knowledge or deep understanding of the three deployed applications or their database layouts.

For obvious reasons, our enforcement technique is only as strong as the provided configuration. If the data controller neglects certain sources or sinks, Fontus will not inspect the data flowing through them and will not enforce data protection checks. Therefore, correct configuration of Fontus is essential for robust enforcement. Fortunately, we observed a large overlap in sources for all three applications. For instance, retrieving HTTP request parameter values is part of the servlet API which builds the foundation for the vast majority of Java web frameworks. Only HTML form parameters were handled differently by each application, but as they all require access to a Servlet object, were trivial to locate. Identifying sinks is more involved as they are dependent on the application architecture. However, this task can be performed relatively easily by identifying places where data leaves the application, such as being read for displaying purposes or module boundaries to identify different processing purposes. Data leaving the application is usually realized via frameworks, e.g., in the form of APIs or the rendering framework and typically have clearly identifiable signatures and annotations. In our experiments, less than ten sinks were needed per application to realize the all of the use cases presented.

Limitations and Future Work Opportunities. The taint engine currently only supports tainting string-like data types. For privacy protection, this limits the amount of data correctly associated to its data subject. For example, Date objects are not tainted by Fontus, but could be used to store dates of birth. Enhancing the underlying taint engine to support tainting additional data types via, e.g., shadow variables, similar to the taint persistence approach described in Section 6.2, without sacrificing too much performance is an interesting challenge.

The presented taint persistence approach relies on parsing SQL queries before handing them to the database driver. Object-relational mapping frameworks are commonly used by large Java applications and dynamically generate SQL Queries by inspecting Java objects. Integrating Fontus with such a framework would avoid costly post-processing as the shadow columns could be directly generated with the query.

In addition, as mentioned earlier, identifying adequate sinks is highly application specific. However, a first step to simplify the configuration process would be to identify common functionality in the JDK that typically transmits data. Fontus already includes sources for reading HTTP request parameters as well as writing to OutputStreams to prevent injection attacks. This default configuration could be extended by a throughout analysis of the Java (EE) standard library to identify common sinks.

9 RELATED WORK

We divide the related work into four areas: dynamic information flow control, taint tracking, GDPR's impact on data processing and attempts to detect mishandling of personal data.

Dynamic Information Flow Control. Dynamic policy enforcement, usually in the form of information flow control (IFC), is an active field of research, with a wide variety of proposed approaches. *Quapla* [31] adds an enforcement engine onto the storage layer by intercepting database queries and rewriting them to add additional restrictions based on provided policies. Such a permission model on the database layer can prevent data leakages but can not prevent policy violations on the application layer. *LEGALEASE* [41] is a full-stack policy enforcement system. It attaches policies to certain attributes, e.g., to prevent their usage for certain purposes. While such policies can prevent some misuses, they are too coarsely grained to support the cases which Fontus enforces. Whether an attribute can be used for a given purpose often varies between users, so a global policy is insufficient.

Resin [54], a PHP runtime preventing security issues via application specific data-flow assertions. It relies on a modified PHP runtime and thus is difficult to deploy securely while also requiring modifications to the application's source code in order to attach labels. IFC approaches for the JVM either add constraints on used language features such as reflection (e.g., *Co-Inflow* [53] or *Aeolus* [7]) or also rely on modifications to the runtime such as *Laminar* [36].

All five of the aforementioned aspects, i.e., encompassing data that is both in flight and persistent, no changes to the application's source code, finely granular policies, no custom runtime as well as support for dynamic language features, are in our opinion crucial for a framework that effectively aids GDPR compliance for real world applications. Therefore, we decided to build Fontus.

Taint Tracking. As a technique to analyze information flows, tainting has been successfully deployed to detect a wide range of issues. Both client [e.g., 24, 32] as well as server-side injection vulnerabilities [e.g., 19, 20] and privacy leaks [e.g., 12] are among its more prominent uses. As the JVM is one of the premier enterprise software environments, different taint-tracking approaches have been proposed for it. The taint tracking part is implemented in a multitude of ways, from including it directly into the JVM [45] over only changing parts of the standard library [18, 19, 26] to bytecode rewriting [3, 28]. We decided to implement our own bytecode rewriting approach since Phosphor is too slow for our use case [20] and the other approaches require changes to either the JVM or the standard library, and thus do not work with standard off-the-shelf JVMs. Persisting taint information in databases has been suggested by Davis and Chen [10], using composite data types not available in every DBMS, whereas our solution, using shadow columns, is fully database agnostic.

Lawful Data Processing under the GDPR. The most visible impact of the GDPR are the so-called cookie notices requesting consent for data processing upon visiting a website. Such notices are consequently well studied, covering aspects of their implementation and compliance with the legal framework [30, 38] and how users interact with them [27, 50]. Several externally quantifiable privacy aspects have been measured in recent years. Among them the presence of client side tracking techniques and third party inclusions [43, 49], the clear communication of data processing purposes, manifested in privacy policies [25], as well as the right to data portability [46]. They all indicate a positive impact of the GDPR. There are still several issues however, for example, unsolicited marketing e-mails upon signing up to websites are still commonplace [23]. Also, data controllers frequently seem to struggle with subject access requests. Martino et al. [29] have shown that a significant number of large organizations perform insufficient identity checks for subject access requests and can be coerced into leaking personal data. This study was replicated in 2021 showing no improvements to the employed processes [11].

One issue in making existing applications with existing data compliant is the lack of association between data and the corresponding data subjects. The approach proposed by Agarwal et al. [1] can infer these relations from existing databases. This can significantly simplify the onboarding for compliance enforcement mechanisms, such as the approach presented in this work.

Preventing GDPR Violations. Wang et al. [52] propose a system that ensures lawful data processing if data and purposes are known before program execution and the application fits into the very limited application model they support. Shastri et al. [42] derive a set of metadata which when attached to stored data can support the compliance process. They then measure the overhead induced by querying and updating these metadata items on a database. While they derive a similar set of metadata compared to ours in Section 4, we believe it to be sufficiently different. Attempts to statically detect GDPR violations [14] differ from our approach in the handling of detected data flows. Unlike static approaches, we can evaluate the lawfulness of the operation based on each user's choices. Additionally, static approaches tend to either over or under approximate data flows in the presence of dynamic code execution, a technique heavily used throughout all demonstrator applications.

10 CONCLUSION

In this work, we proposed framing compliance with data protection laws as an information flow tracking problem. We started by breaking down the legal regulation into twelve distinct software requirements and showed how they can be automatically fulfilled via the definition of a specialized metadata structure. By attaching and propagating this metadata to personal data entering an application, we designed a non-invasive, transparent enforcement mechanism that verifies compliance during data processing.

We implemented our idea in the form of Fontus, which performs taint tracking for the Java Virtual Machine via bytecode rewriting. We were thus able to retrofit existing applications with data protection mechanisms without any code changes at the application level, helping software operators to prevent unintentional data protection violations without costly manual audits.

We demonstrated the practicality of our approach on 7 large open-source Java web applications. We showed that Fontus can robustly enforce data protection policies according to consent preferences provided by the user (i.e., the data subject) during application runtime. Hence, Fontus effectively prevents an application from processing a user's data against their consent.

ACKNOWLEDGMENTS

We would like to thank Jan Niklas Drescher, Louis Bettels and Simon Kurt Petrik for helping making Fontus more robust.

Technische Universität Braunschweig gratefully acknowledges funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and the German Federal Ministry of Education and Research (BMBF) under the project IVAN (16KIS1168). SAP would like to acknowledge support from the German Federal Ministry for Economic Affairs and Climate Action (BMWK) in the project Trade-EVs II (01MV20006A). This work has also received funding from the European Union's Horizon 2020 research and innovation programme under project TESTABLE, grant agreement No 101019206.

REFERENCES

[1] Archita Agarwal, Marilyn George, Aaron Jeyaraj, and Malte Schwarzkopf. 2022. Retrofitting GDPR Compliance onto Legacy Databases. In *VLDB Endow.*

[2] Art. 29 Data Protection Working Party. 2017. Guidelines on the right to data portability (wp242rev.01). https://ec.europa.eu/newsroom/article29/items/611233/en.

[3] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs. In *ACM International Conference on Object Oriented Programming Systems Languages & Applications.*

[4] Bloomberg. 2021. Amazon Gets Record 888 Million Dollar EU Fine Over Data Violations. https://www.bloomberg.com/news/articles/2021-07-30/amazon-given-record-888-million-eu-fine-for-data-privacy-breach. Accessed 08.09.2023.

[5] LLC Broadleaf Commerce. 2022. Broadleaf: Commercial Open Source eCommerce. https://www.broadleafcommerce.com. Accessed 08.09.2023.

[6] LLC Broadleaf Commerce. 2022. MLB Hits a Home Run with Broadleaf. https://www.broadleafcommerce.com/customers/mlb. Accessed 08.09.2023.

[7] Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. 2012. Abstractions for Usable Information Flow Control in Aeolus. In *USENIX Conference on Annual Technical Conference.*

[8] OpenMRS Community. 2022. OpenMRS: Medical Record System. https://openmrs.org. Accessed 08.09.2023.

[9] Council of the European Union and European Parliament. 2016. 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) (Text with EEA relevance).

[10] Benjamin Davis and Hao Chen. 2010. DBTaint: Cross-Application Information Flow Tracking via Databases. In *USENIX Conference on Web Application Development.*

[11] Mariano di Martino, Isaac Meers, Peter Quax, Ken Andries, and Wim Lamotte. 2022. Revisiting Identification Issues in GDPR 'Right Of Access' Policies: A Technical and Longitudinal Analysis. In *Privacy Enhancing Technologies.*

[12] William Enck, Peter Gilbert, Byung Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2019. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX Symposium on Operating Systems Design and Implementation.*

[13] J. S. Fenton. 1974. Memoryless subsystems. *The Computer Journal* 17, 2 (1974).

[14] Pietro Ferrara, Luca Olivieri, and Fausto Spoto. 2018. Tailoring Taint Analysis to GDPR. In *Privacy Technologies and Policy.*

[15] FHIR Foundation. 2022. FHIR: Fast Healthcare Interoperability Resources. https://www.hl7.org/fhir/. Accessed 08.09.2023.

[16] frentix GmbH. 2022. OpenOlat – Infinite Learning. https://www.openolat.com. Accessed 21.01.2022.

[17] Gatling Corp. 2022. Gatling. https://gatling.io. Accessed 08.09.2023.

[18] Vivek Haldar, Deepak Chandra, and Michael Franz. 2005. Dynamic Taint Propagation for Java. In *Annual Computer Security Applications Conference.*

[19] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. 2006. Using Positive Tainting and Syntax-aware Evaluation to Counter SQL Injection Attacks. In *ACM International Symposium on Foundations of Software Engineering.*

[20] Katherine Hough, Gebrehiwet Welearegai, Christian Hammer, and Jonathan Bell. 2020. Revealing Injection Vulnerabilities by Leveraging Existing Tests. In *ACM/IEEE International Conference on Software Engineering.*

[21] Informatici senza Frontiere. 2022. Open Hospital: Software EMR HIS open source. https://www.open-hospital.org/. Accessed 22.04.2023.

[22] JForum Team. 2022. JForum. https://jforum.net/. Accessed 08.09.2023.

[23] Karel Kubíček, Jakob Merane, Carlos Cotrini, Alexander Stremitzer, Stefan Bechtold, and David Basin. 2022. Checking Websites' GDPR Consent Compliance for Marketing Emails. In *Privacy Enhancing Technologies.*

[24] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *ACM Conference on Computer and Communications Security.*

[25] Thomas Linden, Rishabh Khandelwal, Hamza Harkous, and Kassem Fawaz. 2020. The Privacy Policy Landscape After the GDPR. In *Privacy Enhancing Technologies.*

[26] Florian D Loch, Martin Johns, Martin Hecker, Martin Mohr, and Gregor Snelting. 2020. Hybrid Taint Analysis for Java EE. In *ACM Symposium on Applied Computing.*

[27] Dominique Machuletz and Rainer Böhme. 2020. Multiple Purposes, Multiple Problems: A User Study of Consent Dialogs after GDPR. In *Privacy Enhancing Technologies.*

[28] Michael Martin, Benjamin Livshits, and Monica Lam. 2006. *SecuriFly: Runtime Protection and Recovery from Web Application Vulnerabilities.* Technical Report. Stanford University.

[29] Mariano Di Martino, Pieter Robyns, Winnie Weyts, Peter Quax, Wim Lamotte, and Ken Andries. 2019. Personal Information Leakage by Abusing the GDPR 'Right of Access'. In *USENIX Security Symposium.*

[30] Célestin Matte, Nataliia Bielova, and Cristiana Santos. 2020. Do Cookie Banners Respect my Choice? : Measuring Legal Compliance of Banners from IAB Europe's Transparency and Consent Framework. In *IEEE Symposium on Security and Privacy.*

[31] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. 2017. Qapla: Policy compliance for database-backed systems. In *USENIX Security Symposium.*

[32] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting. In *Network and Distributed System Security Symposium.*

[33] Microsoft. 2023. Playwright. https://github.com/microsoft/playwright. Accessed 08.09.2023.

[34] Reuters. 2021. WhatsApp fined a record 225 mln euro by Ireland over privacy. https://www.reuters.com/technology/irish-data-privacy-watchdog-fines-whatsapp-225-mln-euros-2021-09-02/. Accessed 08.09.2023.

[35] Reuters. 2022. Google hit with 150 million euro French fine for cookie breaches. https://www.cnbc.com/2022/01/06/google-hit-with-150-million-euro-french-fine-for-cookie-breaches.html. Accessed 08.09.2023.

[36] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. 2009. Laminar: practical fine-grained decentralized information flow control. In *ACM Conference on Programming Language Design and Implementation.*

[37] Marlene Saemann, Daniel Theis, Tobias Urban, and Martin Degeling. 2022. Investigating GDPR Fines in the Light of Data Flows. In *Privacy Enhancing Technologies.*

[38] Iskander Sanchez-Rola, Matteo Dell'Amico, Platon Kotzias, Davide Balzarotti, Leyla Bilge, Pierre-Antoine Vervier, and Igor Santos. 2019. Can I Opt Out Yet? GDPR and the Global Illusion of Cookie Control. In *ACM Asia Conference on Computer and Communications Security.*

[39] SAP. 2023. CAP Bookstore. https://github.com/SAP-samples/cloud-cap-samples-java. Accessed 08.09.2023.

[40] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy.*

[41] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Y. Tsai, and Jeannette M. Wing. 2014. Bootstrapping Privacy Compliance in Big Data Systems. In *IEEE Symposium on Security and Privacy.*

[42] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. 2020. Understanding and Benchmarking the Impact of GDPR on Database Systems. In *VLDB Endow.*

[43] Jannick Kirk Sørensen and Sokol Kosta. 2019. Before and After GDPR: The Changes in Third Party Presence at Public and Private European Websites. In *International World Wide Web Conference.*

[44] Sarah Spiekermann. 2012. The Challenges of Privacy by Design. *Commun. ACM.*

[45] Bruno Crispo Srijith K. Nair, Patrick N.D. Simpson and Andrew S. Tanenbaum. 2008. *IR-CS-045: Trishul: A Policy Enforcement Architecture for Java Virtual Machines.* Technical Report. Vrije Universiteit.

[46] Emmanuel Syrmoudis, Stefan Mager, Sophie Kuebler-Wachendorff, Paul Pizzinini, Jens Grossklags, and Johann Kranz. 2021. Data Portability between Online Services: An Empirical Analysis on the Effectiveness of GDPR Art. 20. In *Privacy Enhancing Technologies.*

[47] The Spring PetClinic Community. 2022. Spring PetClinic. https://spring-petclinic.github.io. Accessed 08.09.2023.

[48] University of Applied Sciences Mannheim. 2022. HSMA-CTT. https://github.com/informatik-mannheim/HSMA-CTT. Accessed 08.09.2023.

[49] Tobias Urban, Dennis Tatang, Martin Degeling, Thorsten Holz, and Norbert Pohlmann. 2020. Measuring the Impact of the GDPR on Data Sharing in Ad Networks. In *ACM Asia Conference on Computer and Communications Security.*

[50] Christine Utz, Martin Degeling, Sascha Fahl, Florian Schaub, and Thorsten Holz. 2019. (Un)informed Consent: Studying GDPR Consent Notices in the Field. In *ACM Conference on Computer and Communications Security.*

[51] VMware, Inc. 2022. Spring. https://spring.io. Accessed 08.09.2023.

[52] Lun Wang, Usmann Khan, Joseph P. Near, Qi Pang, Jithendaraa Subramanian, Neel Somani, Peng Gao, Andrew Low, and Dawn Song. 2022. PrivGuard: Privacy Regulation Compliance Made Easier. In *USENIX Security Symposium.*

[53] Jian Xiang and Stephen Chong. 2021. Co-Inflow: Coarse-grained Information Flow Control for Java-like Languages. In *IEEE Symposium on Security and Privacy.*

[54] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2009. Improving Application Security with Data Flow Assertions. In *ACM SIGOPS Symposium on Operating Systems Principles.*

[55] Wei You, Bin Liang, Wenchang Shi, Peng Wang, and Xiangyu Zhang. 2017. Taintman: An art-compatible dynamic taint analysis framework on unmodified and non-rooted android devices. *IEEE Transactions on Dependable and Secure Computing.*