Making Web Applications GDPR Compliant: A Comparative Evaluation of GDPR-Enforcement Frameworks

Felix Kalinowski
Ruhr University Bochum
felix.kalinowski@ruhr-uni-bochum.de

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

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

Veelasha Moonsamy
Ruhr University Bochum
email@veelasha.org

Abstract

The introduction of the General Data Protection Regulation (GDPR) in 2018 marked a pivotal moment in the evolution of data protection within the European Union (EU). Consequently, companies have since been legally obliged to respect users' privacy, and, if found to be in violation, risk incurring fines. While this regulatory change greatly benefits users, software developers, on the other hand, face a tremendous challenge to make their applications compliant, creating a gap between legal requirements and effective software development. Several solutions have been proposed to bridge the gap for web application developers. However, it is unclear to what extent they fulfill the requirements laid out by the GDPR. In this work, we look at three frameworks that aim to aid compliance for web applications. To efficiently assess them, we propose a methodology and several benchmarks to evaluate and compare the frameworks. From the GDPR, we have derived a set of requirements that do not entail institutional changes but have technical implications for software. Leveraging these requirements, we evaluate both the proposed solutions' enforcement capabilities and computational overhead. Our comparison shows that each framework can, if configured correctly, enforce a different subset of GDPR requirements. Finally, based on the insights gained, we provide recommendations for the community on how to make further progress on operationalizing the GDPR.

Keywords

GDPR enforcement, privacy, web applications

1 Introduction

Digitalization is nowadays omnipresent in every aspect of people's lives: from sharing electronic health records across the EU [2] to having a digital identity in addition to a physical identity card [3] or online classes during a pandemic [11]. As a result, personal data is often collected through web applications that provide users with the mentioned services, including social networks, online shops, and entertainment services. In turn, web applications collect and process a wide range of personal data, building up detailed profiles of users to predict their behaviors. This leads to technology companies using personal data increasingly focusing on advertising [1], making adverts more targeted to individual users to increase click-through rates and, consequently, their revenue. A recent report revealed the fine-grained categorization of users for targeted advertising by data brokers who focus on monetizing the users' personal data [10]. In addition to advertising, personal data is also used to influence users' choices, for example, during elections [22, 26]. Those examples show that personal data is used to sway users' purchasing behaviours online and to actively influence democratic elections.

To protect the citizens' data and minimize the influence of companies on users' individual choices, the EU has passed a regulation, the General Data Protection Regulation (GDPR) [6]. The GDPR is based on seven principles: *Lawfulness*, *fairness and transparency*, *Purpose limitation*, *Data minimisation*, *Accuracy*, *Storage limitation*, *Integrity and confidentiality*, and *Accountability* (Art. 5 GDPR). Since the GDPR applies to any entity processing personal data of individuals in the EU, web application developers and operators are responsible for ensuring that their systems comply with its requirements. This task presents several challenges: Any web application, including those developed before the introduction of GDPR, must be adapted at every point in the codebase where personal data is processed. Furthermore, developers need to fully understand the requirements introduced by the GDPR to implement them correctly. This can lead to issues, as they are usually not trained in the legal aspects of such regulations. In addition, the GDPR requires specific functionalities for user-based inputs such as data manipulation (deletion, modification) or consent-based data processing. Subsequently, developers must ensure that these functionalities are represented in the application. This leads to the questions: *how to ensure that a web application is GDPR-compliant?* and *how to aid developers in enforcing the requirements?* One way to address these issues is to separate the application codebase from the GDPR requirements by adding a dedicated layer for GDPR-compliant data processing. This layer would act as an intermediary between the application and the underlying data management systems, ensuring that all data processing activities, such as consent management, data access, or erasure requests, comply with the GDPR.

In this paper, we systematically compare and evaluate the aforementioned layer in the form of frameworks [7, 15, 19] that allow their integration into web applications. Based on a provided policy, these frameworks monitor data processing functions and determine whether a data processing operation is permitted and ultimately suppress or allow it. In these frameworks, user data is tagged with

Proceedings on Privacy Enhancing Technologies 2025(4), 777–794

https://doi.org/10.56553/popets-2025-0157

unique identifiers to determine whether a particular user's data can be processed on a legal basis, such as the user's consent. Such type of data protection applies not only to data going in and out of the web application but also to any access to the database where the data is stored.

In this work, we make the following contributions:

(i) We propose a list of 16 technically enforceable requirements derived from the GDPR with the help of a legal expert.
(ii) We propose a methodology and benchmarks to measure the GDPR enforcement capabilities and performance of a GDPR-enforcing framework for web applications.
(iii) We evaluate our benchmarks on three state-of-the-art frameworks.
(iv) We derive recommendations for the community based on our findings, to provide better tooling for operationalizing GDPR.
(v) In the spirit of open science and to help with reproducibility, we plan to make our evaluation setup and source code available upon paper acceptance.

Disclose of use of AI-based tools for writing assistance: we used the services of DeepL and Grammarly for translations, spelling, and grammar checks on text across the paper.

2 Background

In this section, we briefly introduce the relevant GDPR articles about the processing of personal data and explain the concept of personal data tainting for information flow tracking. Unless noted otherwise, all references to articles (Art.) in this work refer to the GDPR.

2.1 General Data Protection Regulation (GDPR) & Personal Data

The GDPR is a comprehensive data protection law enacted by the European Union to safeguard individuals' privacy and personal data and protect their fundamental rights. It establishes stringent requirements for organizations that handle personal data, emphasizing transparency, security, and individuals' rights. The GDPR establishes the core principles and requirements for the lawful processing of personal data; this includes principles such as transparency, or data minimization (Art. 5). Additionally, it defines the conditions under which personal data processing is considered lawful (Art. 6, 9), outlining various legal bases such as consent or legal obligation (Art. 6). These provisions ensure that data processing activities are conducted responsibly and in accordance with individuals' rights. The GDPR also outlines different rights of data subjects, ensuring individuals have control over their personal data, such as access to their data, rectification of inaccuracies, erasure (the *right to be forgotten*), or data portability (Art. 15, 18, 20, 21). These rights empower individuals to manage their personal data and ensure transparency and accountability from data controllers. Additionally, Art. 30 mandates that data controllers and processors maintain records of their processing activities. These records must include information about the categories of personal data processed, the purposes of processing, and the data's recipients. While the GDPR sets out multiple provisions for data processing, it intentionally remains broad and does not provide detailed guidance on how companies and developers should operationalize these provisions. This flexibility ensures the regulation can be widely applied across various contexts, but also creates a gap between legislation and practical implementation.

```
"Alice": {
    "age": 25, "taint": "ef29755a",
    "blood_pressure": 120, "taint": "e37311c3",
    "weight": 70, "taint": "b83a7ba6T"
}
```

Listing 1: Example of Taints on Personal User Data.

2.2 Personal Data Tainting for Information Flow Tracking

One approach to support GDPR compliance is to be able to track data flowing through a web application. A common technique used for such purpose is *data tainting* [25]. This allows metadata to be attached to the original data. This means, for example, that data can be marked as personal data belonging to a specific data subject and a framework can enforce requirements on the data, like the right to be forgotten or the right to access the data. For example, consider a web application that processes personal user data, such as health information, in a web form – as shown in Listing 1. The data subject is Alice, and several health-related data items belong to her. As per the GDPR, the data subject has the right to access this data and the right to be forgotten. With data tainting, the data can be marked as belonging to Alice, and the framework can enforce these GDPR requirements on the data. The *taint* fields in Listing 1 contain unique identifiers for Alice's personal data. Therefore, if Alice requests her data to be deleted, the data processor can query all the data tainted as belonging to Alice from the database and delete it on her behalf. Additionally, the taints allow for tracking data throughout the application.

3 GDPR-Enforcement Frameworks

In this section, we first outline the criteria used to select the three frameworks, followed by a comprehensive description of each one.

3.1 Framework Selection

We used the following criteria to select the frameworks: 1) **GDPR Enforcement Capability**: As our primary objective is to evaluate GDPR compliance, the framework should provide a mechanism for enforcing GDPR properties. 2) **Open Source**: The framework should be open source and freely available, allowing us to evaluate without restrictions. 3) **Support for Web Application**: The framework should target web applications, as this is the focus of our study. 4) **State-of-the-Art Technology**: The framework should be built on commonly used technology (e.g., Spring Boot, Express.js, Flask, etc.). This ensures the framework is current and can be used for modern web applications. We searched the last 5 years of top A/A* security and privacy conferences (S&P, NDSS, PETS, ESORICS, EuroS&P) for GDPR enforcement tools. We found 5 tools, out of which we discarded 2 not matching our criteria. This left us with

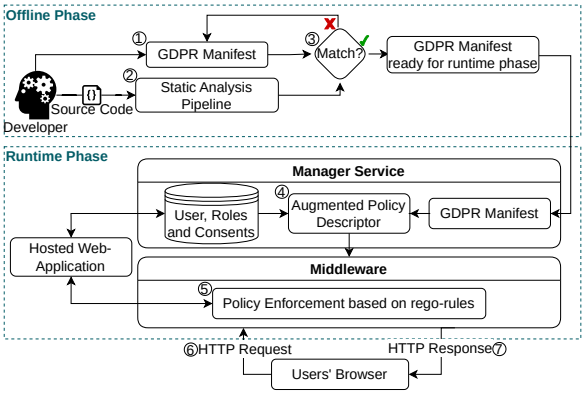


Figure 1: Overview of RuleKeeper's Architecture [7].

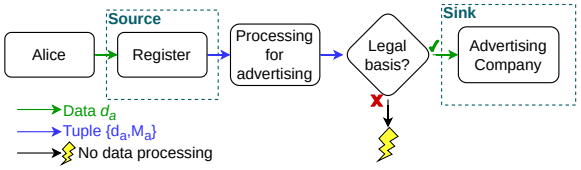


Figure 2: Overview of Fontus Pipeline [19].

3 frameworks for comparative evaluation. However, we acknowledge that relevant work may also exist in other subfields, such as software engineering, and this represents a potential limitation in our selection process. Nonetheless, our goal was to evaluate mature, peer-reviewed frameworks within the privacy and security research community, since venues like ICSE (software engineering) or VLDB (database systems) have less visibility in the targeted community. These criteria led us to the selection of the following state-of-the-art frameworks, which we describe in further detail below: RuleKeeper [7], Fontus [19], and GDPR-MFOTL [15].

3.2 RuleKeeper [7]

The RuleKeeper framework by Ferreira et al. [7] is the sole framework among the three that not only claims to enforce a set of GDPR requirements but also assists developers in creating compliant applications. Its objective is to prevent developers from accidentally introducing GDPR violations in the first place. As shown in Figure 1, RuleKeeper is split into two phases: The *Offline Phase* helps developers to set up their web application inside RuleKeeper, while the *Runtime Phase* enforces various aspects of the GDPR, like lawful data processing or purpose limitation, on the hosted application. As a running example, we consider a simple web application for which users can register with a username, an email, and a password, choose from a set of interests, and receive blog posts for their chosen interests.

To set up their web application in RuleKeeper, developers must ① create a *developer's manifest* that categorizes all processed data into personal and non-personal. The developer's manifest also specifies allowed operations, their legal bases, and corresponding data items permitted to be collected for each operation. In our example, personal data includes user interests and email. For simplicity, we treat the username and password as non-personal data, even though they would typically be considered personal in real-world scenarios. Operations like receiving blog posts and sending marketing emails are mapped to endpoints, such as /blogposts for user interests and /send-news-mails for the email. Developers must detail the processed data and legal basis in the developer's manifest, with the legal basis represented as a string, e.g., *contract* or *legal_obligation*, without validity checks. Next, ② developers run their web application's source code through a static analysis pipeline to generate a *static analysis manifest*. The two manifests, the developer's manifest and the static analysis manifest, are then ③ compared for discrepancies. If they do not match, developers must update either the source code or the developer's manifest. The developer's manifest is ready for the *Runtime Phase* if they match. Data collection occurs in the offline phase, where relevant data is identified and linked to the corresponding endpoint via static analysis. This aspect is implicitly assumed during the runtime phase, as the data to be collected has already been predefined. The runtime phase starts with the *Manager Service*, which holds the developer's manifest and a database containing users, user roles, and consent choices. These consent choices and the developer's manifest are ④ merged into an *Augmented Policy Descriptor (ADP)* policy file using a Domain Specific Language (DSL). RuleKeeper also features a ⑤ middleware that enforces the ADP on the hosted web application. Therefore, when a user sends a ⑥ HTTP request to an endpoint of the hosted web application, RuleKeeper processes the data according to the current ADP and ⑦ returns the HTTP response. For instance, if a user queries the /blogposts endpoint and consents to the processing of their interests, they receive the relevant posts, or an empty response otherwise.

Implementation. RuleKeeper requires applications to use the MERN stack: MongoDB, Express, React, and Node.js. This means that the web application, which should be hosted in RuleKeeper, must also use MongoDB as its database system and be written with the Node.js framework Express. React must be used for the frontend. However, we did not build a frontend for our evaluation. To enforce the APD at runtime, RuleKeeper utilizes the open-source tool Open Policy Agent (OPA) [17], which uses a DSL called Rego, which is used to write policies. The authors of RuleKeeper provide multiple Rego policies used to enforce different aspects of the GDPR, such as purpose limitation or data minimization. To enforce those aspects during runtime, RuleKeeper relies on the ADP, which contains information such as the data allowed to be processed for each endpoint, and combines it with runtime information. For example, whenever a user queries an endpoint, RuleKeeper first invokes the access control Rego policy to be sure that the user is allowed to use the endpoint. Afterward, if the endpoint queries the MongoDB collection holding personal user data, RuleKeeper invokes another Rego policy to check whether the endpoint is allowed to process the specific data items and if there is a valid legal basis to do so.

3.3 Fontus [19]

Fontus, depicted in Figure 2 and proposed by Klein et al. [19], is based on a dynamic data-flow tracking approach. It comprises two

main components: *Sources* and *Sinks*. Every part of the web application where user data is introduced is called a *source*, and where user data is processed or leaves the application is called a *sink*. Continuing with the running example introduced in RuleKeeper (Section 3.2), consider a user named Alice registering for the blog application. She selects her interests and provides her email address, corresponding to data items d_a. As this information is entered through the registration endpoint, that endpoint functions as a source in the system. Fontus now appends metadata M_a to Alice's data, forming the tuple $\{d_a, M_a\}$. This metadata M holds multiple pieces of information as defined in

$$M = \langle Q, f, l, S, i, p, r \rangle \tag{1}$$

where Q corresponds to a purpose-recipient pair, f describes the time period during which processing is allowed, l is the required protection level for data d, S is a set of data subjects associated with data d, identifiable by a pseudonym or identifier, i defines a unique identifier, p indicates whether data qualifies for portability with $p \in \{0, 1\}$ and r defines whether processing of data d is restricted, with $r \in \{0, 1\}$. This metadata is attached to each personal data item being processed within Fontus. Returning to our example, consider an advertising company that wants to use Alice's data for marketing purposes. In this case, the point where personal data is transmitted to the advertising company is considered a sink, marked with the purpose and recipient pair Q_x. Before invoking the functionality, i.e., transmitting Alice's data to the company, Fontus checks if M_a contains Q_x. If not, Fontus suppresses the processing. However, Fontus cannot assess whether there is a valid basis and relies on the developer to label personal data correctly. With the Q_x pair, Fontus can also determine which third party receives which data for which purpose. However, this is only the case for server-side third parties.

Implementation. The authors of Fontus implement it as a taint-tracking engine to enforce GDPR compliance at runtime with minimal changes to the hosted Java web application. It rewrites the byte code of string-like classes, adding fields for the original class reference and associated metadata within "taint ranges" that identify subsequences of strings with personal data. This ensures granular tainting down to individual characters. When tainted data is stored in the database, Fontus also stores the associated taint metadata. This ensures that when the data is retrieved later, taint tracking can continue correctly. Fontus's taint metadata contains runtime information, such as the logged-in user. Fontus relies on a so-called *Taint Handler* to enhance the taint metadata with such application-specific runtime data. The *Taint Handler* generates taint values for new personal data entering the application (at a source) and checks taint values for data about to be processed (at a sink). It retrieves personal data and the current user's context (e.g., the username or user id) at a source, generates a metadata tuple, and attaches the corresponding taint value. At a sink, the *Taint Handler* hooks into the application, retrieves the personal data about to be processed and its metadata tuple, and uses the metadata tuple to guide the application on how to process the data. Developers must specify the necessary information in an XML file, including the purpose and recipient pairs. They must also map the methods through which personal data enters the application to the *Taint Handler* methods for sources. Furthermore, they must map the functions that process personal data (i.e., sink functions) to their corresponding *Taint Handler* methods.

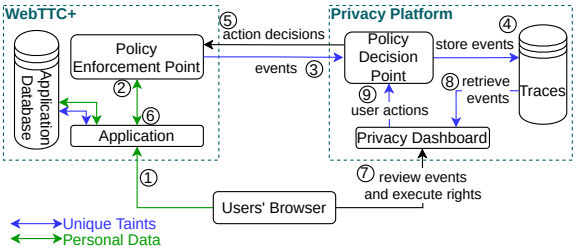


Figure 3: Overview of GDPR-MFOTL Architecture [15].

3.4 GDPR-MFOTL [15]

Hublet et al. [15] built their framework, similar to Fontus, based on a data tainting approach. For each data item to be processed, a unique identifier called *unique taint* is attached, allowing the data item to be tracked throughout the application and influence further processing. The general architecture of GDPR-MFOTL is shown in Figure 3.

It is split into two main components, a *Privacy Platform* and *WebTTC+*. The WebTTC+ component hosts the web application, the corresponding database, and the *Policy Enforcement Point*. Consider the previously introduced running example (see Section 3.2). Whenever a user ① interacts with the hosted web application, and personal data is processed, WebTTC+ ② invokes the policy enforcement point. This component's purpose is to deny or allow further processing of personal data. The policy enforcement point then emits an event ③, for example, the user wanting to access blog posts based on their interest, to the *Policy Decision Point*, which resides inside the Privacy Platform. Like RuleKeeper, GDPR-MFOTL utilizes its DSL to translate (parts of) the GDPR into a policy used for enforcement. The policy is written in Metric First-Order Temporal Logic (MFOTL) [14] using an enforcement tool called EnfPoly [14]. This policy is used in the policy decision point to decide whether processing is allowed. After storing ④ the event in a separate traces database, the decision is then returned ⑤ to the policy enforcement point, which enforces the decision on the application ⑥. Another component of the privacy platform is the *Privacy Dashboard*, which users can access to review events and exercise their rights ⑦. The privacy dashboard displays ⑧ previously emitted events that concerned the users' personal data. Whenever the user executes an action (e.g., providing consent for a specific purpose), the action is provided ⑨ to the policy decision point to influence further processing.

Implementation. GDPR-MFOTL targets Python applications written in Flask. Additionally, the authors introduce a programming language called PythonTTC (a subset of Python), also developed by the authors [16]. To run a web application in GDPR-MFOTL, the developers must first translate it into PythonTTC. Additionally, every time personal data is processed, developers have to

invoke the built-in *check* functions to query WebTTC+ for the lawfulness of processing. Besides that, data tainting and tracking are automated by GDPR-MFOTL. To enforce GDPR requirements at runtime, GDPR-MFOTL uses the aforementioned MFOTL, which consists of multiple *predicates* logically combined into formulas. Equation (2) shows the formula for purpose-based processing.

$$\begin{aligned}\varphi_{Purp} = \exists\, prp,\ ut,\ ds,\ sp.\\ \text{Use}(prp,\ ut)\ \wedge \blacklozenge\text{Collect}(ds,\ ut,\ sp)\\ \wedge\ \neg((\neg\text{DSRevoke}(ds, prp, ut)\ \hat{S}\ \text{DSConsent}(ds,\ prp,\ ut))\\ \vee\ (\exists grd.\ \blacklozenge\text{LegalGround}(grd,\ ut,\ sp)))\end{aligned} \tag{2}$$

The predicate *Use* indicates the application intends to utilize a specific data item for a defined purpose. *Collect* refers to gathering data with a unique taint. *DSConsent* signifies that the data subject consents to the data item, while *DSRevoke* denotes the revocation of that consent. Finally, *LegalGround* permits developers to process data based on a legal basis other than consent. Combining the predicates, the formula states that there exists a purpose *prp*, a unique taint *ut*, a data subject *ds*, and a special-data-flag *sp*. The *ut* is being used for *prp* and the *ut* has been collected from the *ds* with *sp* and neither the *ds* has given consent to process *ut* for *prp* nor revoked the consent since then nor has the application claimed a legal ground *grd* for *ut* with *sp*. We refer to the original paper [15] for the complete list of predicates and formulae. To make decisions based on previous events, like retrieving consent in the past, the policy decision point stored the events in a traces database, as explained previously. The combination of prior traces and decision making is stored in a decision log as shown in Listing 2. We also found a discrepancy between the MFOTL formulae introduced in the paper and the actual implementation. In addition to the arguments defined in the paper, the implementation features the application's name in the different functions (e.g., *BreezeBlogs* for our running example)

Following our running example, a user registers with the BreezeBlogs application, sets their interests, and wants to access the corresponding blog posts. During the registration, the predicate *Collect* is called (line 1), stating that the user's interests have been collected for the application *BreezeBlogs* with the unique taint *1234*. This is acknowledged by the policy decision point (line 2). Afterward, the application *BreezeBlogs* wants to *Use* the data with unique taint *1234* for the purpose *view_posts* (line 3). This is suppressed by the policy decision point (line 4) because the data subject neither provided consent nor did the application claim a legal ground for processing.

```
1 @1715596172 Collect (1, "BreezeBlogs", "1234", "False") ; // taint
  ↪  "1234" corresponds to the user's interests
2 [Enforcer] OK.
3 @1715596172 Use ("BreezeBlogs", "view_posts", "1234") ;
4 [Enforcer] Suppress: Use("BreezeBlogs","view_posts","1234")
5 @1715596172 DSConsent (1, "BreezeBlogs", "view_posts", "1234") ;
6 [Enforcer] OK.
7 @1715596172 Use ("BreezeBlogs", "view_posts", "1234") ;
8 [Enforcer] OK.
```

Listing 2: EnfPoly Enforcement Example.

Afterwards, the data subject consents to the processing (line 5), and the usage is no longer suppressed (line 8). Using this approach, the framework can make decisions based on previous events and enforce aspects of the GDPR at runtime.

4 Our Methodology

In this section, we describe the methodology used to conduct our experiments and elaborate further on the test application we developed for our experimental evaluation.

4.1 Evaluation Criteria

We now detail our evaluation criteria to compare the three frameworks. The evaluation is divided into three parts: GDPR coverage, runtime overhead, and storage overhead.

GDPR Coverage. To identify which GDPR requirements to include in our evaluation, we collaborated with a legal expert specialized in data protection law. Rather than attempting to operationalize the entire regulation, we followed a targeted selection process based on clearly defined criteria. Specifically, we focused on requirements that (i) can be measurable, verifiable technical controls or features within software systems, (ii) can be assessed through observable behavior - such as API outputs or data flows -, documentation, or interfaces, and (iii) do not rely on subjective legal interpretation or organizational context. This process involved reviewing the GDPR from a technical perspective, emphasizing actionable provisions that are potentially suitable for automated or semi-automated analysis. Requirements that depend heavily on contextual, legal or organizational factors were excluded, as they cannot be reliably evaluated without human judgment. The resulting list of requirements, albeit being a subset of the GDPR, covers a broad range of obligations relevant to technical system design and behavior, and serves as the basis for our framework evaluation. For each GDPR requirement, we evaluate three stages, following a hierarchical approach:

Stage 1: Supported? Does the paper claim to support the requirement?

Stage 2: Implemented? Is the requirement implemented?

Stage 3: Enforced? Can the framework enforce the requirement for the test application?

In the first stage, we determine if the authors of the three frameworks claim to support the different GDPR requirements. If not, we do not need to evaluate the requirement further. If the authors claim to support the requirement, we evaluate, in the second stage, whether the requirement is implemented in the framework by investigating the framework's source code. If the requirement is implemented, we evaluate whether the framework enforces it on the hosted application in the third stage by conducting tests. For example, if one of the papers claims to support the right to erasure, we look at the framework's source code to determine if the right to erasure is implemented. If so, we determine if the user can delete their data from our test application. This way, we can determine if the frameworks enforce the different GDPR requirements both theoretically and in practice.

Runtime Overhead. In addition to GDPR coverage, we also want to determine how much runtime overhead the three frameworks

introduce on top of our test application. To measure the runtime overhead, we use the Locust load evaluation tool [18]. With Locust, we run several benchmarks for our test application, either standalone or hosted inside the frameworks. Afterwards, we compare the response times of the different benchmarks to determine the overhead added by the frameworks. Additionally, as web applications are typically used by multiple users simultaneously, we also evaluate the overhead in a multi-user context. However, since GDPR-MFOTL does not support multi-user execution at the time of writing, we thus only evaluate the overhead in a single-user context. Hence, our evaluation consists of the following two benchmarks: (1) Single User: In this benchmark, we evaluate the overhead added in a single-user context, where no concurrency is involved. (2) Multi User: Here, we determine the overhead added when 50, 100, and 500 users access the application concurrently. Moreover, in addition to evaluating the web application within the frameworks with multiple users, we also vary the number of users who gave consent ((i) no user gave consent, (ii) half of the users gave consent, and (iii) every user gave consent) for the purposes of marketing and viewing the blog posts. This way, we measure the baseline, the runtime overhead added by the frameworks, and the impact of GDPR enforcement on the overhead. For example, response times may be reduced if the framework suppresses certain data processing operations due to a lack of user consent. We give each framework a budget of 3+10 minutes and execute the same task in a loop to measure the average response time. The first three minutes are discarded to allow the system to reach a steady state, as typically users interact with a system once all the initialization routines have been completed. We then measure the response times over the remaining 10 minutes. All benchmarks are executed in a virtual machine with 12 cores and 32 GB of RAM running Ubuntu 24.04. We monitor the system's resource utilization during the benchmarks to ensure that excessive CPU usage does not affect the results. We refer to Section B for additional runtime overhead measurements.

Storage Overhead. For this evaluation, we set up our test application with and without the frameworks with varying numbers of users (50, 100, 500). We then determine all places where the frameworks demand additional storage (excluding the source code and log files) and calculate the storage used by our test application with and without the frameworks. For each of the storage backends (MongoDB for RuleKeeper, MySQL for Fontus, and SQLite/QuestDB for GDPR-MFOTL), we use the default configurations provided by the authors. Additionally, we use the built-in tools of the storage backends to release allocated storage from previous evaluation runs. Contrary to Fontus and GDPR-MFOTL, as RuleKeeper does not rely on data tainting, we set up the databases without running the framework and then use MongoDB's internal tools to calculate the data size. After registering the desired number of users, we then calculate the allocated size. We have to note here that both MySQL and QuestDB allocate more storage than the required amount to hold the data, and we did not find a reliable way to determine the exact amount, leading to a higher overhead due to pre-allocation.

Original Evaluation & Comparability. In the original papers, all of the three frameworks were evaluated differently: RuleKeeper assessed server/client latency, throughput, and CPU/memory efficiency for up to 64 users; GDPR-MFOTL measured latency for 100 requests with one user; and Fontus evaluated latency up to 100 users and includes storage overhead. Notably, only Fontus evaluated storage overhead, which we also consider. Beyond runtime/storage overhead, the frameworks were evaluated initially for their ability to enforce GDPR requirements on web applications. We also unify this by evaluating the frameworks against the same web application and using a more granular legal analysis. We argue that the frameworks can be meaningfully compared because they all aim to enforce GDPR compliance in web applications. Despite differences in technologies and ecosystems, their core goal, privacy and data protection under GDPR, remains the same. This comparison offers insights into how each framework addresses the same challenge, highlighting performance trade-offs and overheads.

4.2 Test Application: `BreezeBlogs`

To evaluate the three introduced frameworks, we have developed *BreezeBlogs*, a simple web application that implements the idea of a blog aggregator based on different interests. Users who register for BreezeBlogs select their interests from various options (such as travel, food, or technology). The user can then view blog posts based on their chosen interests. BreezeBlogs needs at least one database to store the user information and the blog posts. However, the size and structure of the database vary from framework to framework. In addition to the blog functionality, BreezeBlogs has a newsletter feature that sends marketing emails to all registered users' email addresses to mock a marketing campaign. BreezeBlogs is designed with different GDPR requirements in mind. With the user's interests and email address, we have two personal data items processed for different use cases. The interests are used to fetch the blog posts for the user, while the email address is used to send marketing emails. This allows us to test GDPR compliance in different scenarios. With the blog posts and marketing functionality, we can test for the lawfulness of processing and purpose limitation. In addition, we can test for requirements like data minimization and different data subject rights by having independent types of personal data.

Since each framework targets a different programming language, we developed BreezeBlogs in Python, Java, and JavaScript, respectively. BreezeBlogs implements the following HTTP/API endpoints: 1) **POST /register** Registers the user in the database. 2) **POST /login** Logs the user in. 3) **GET /interests** Fetches and returns the current user's interests. 4) **GET /blog-posts** Accesses the current user's interest, then, for the user's interests, fetches 10 blog posts with 1024 characters each and returns them. 5) **POST /send-news–mails** Queries *all* users' usernames and email addresses from the database and joins them together into a single string.

We chose the endpoint functionality so that we have an endpoint processing a lot of non-user-specific data (/blog-posts), an endpoint processing user-specific data from multiple data subjects at once (/send-news-mails), and an endpoint processing user-specific data from a single data subject (/interests). This way, we can determine the impact of the enforcement on different types of data processing.

5 Evaluation

In this section, we evaluate the frameworks' GDPR coverage and compare them based on the runtime and storage overheads they introduce to BreezeBlogs.

Table 1: GDPR Coverage per Framework.

GDPR Requirement		RuleKeeper	Fontus	GDPR-MFOTL
Purpose Limitation	Art. 5(1)(b)	●	●	○
Data Minimization	Art. 5(1)(c)	●	○	○
Storage Limitation	Art. 5(1)(e)	○	◐	○
Integrity and Confidentiality	Art. 5(1)(f)	●*	○	○
Lawfulness of Processing†	Art. 6(a-f)	●	●	●
Consent Handling	Art. 7(1-4)	◐	◐	●
Special Data	Art. 9	○	◐	◐
Transparency	Art. 12	○	○	●
Information to be provided (Total count = 12)	Art. 13	1/12	2/12	1/12
Data Subject Rights				
- Right to Access	Art. 15	○	◐	◐
- Right to Rectification	Art. 16	○	◐	●
- Right to Erasure	Art. 17	○	◐	◐
- Right to Restriction of Processing	Art. 18	○	◐	◐
- Right to Data Portability	Art. 20	○	◐	●
- Right to Object	Art. 21	○	◐	◐
Records of processing activities (Total count = 7)	Art. 30	0/7	2/7	1/7

○: Not Supported, not implemented, not enforced; ◐: Supported, not implemented, not enforced; ●: Supported, implemented, enforced; *: Confidentiality is implemented in RuleKeeper as access control, integrity is not supported; †: Although it is the only requirement enforced by each framework, correct human assessment for the legal basis is still required.

5.1 GDPR Coverage

Table 1 provides a comparative evaluation of the GDPR coverage for all three frameworks.

Purpose Limitation. Purpose limitation requires personal data to be collected for specified, explicit, and legitimate purposes and not further processed in a manner incompatible with those (Art. 5(1)(b)). This principle is central to the GDPR: it anchors many other obligations, such as data minimization, lawful basis, and user consent. In practice, purpose limitation ensures that personal data collected for one use cannot be repurposed for unrelated uses (e.g., marketing) without a legal basis. We assessed purpose limitation by defining two different purposes in BreezeBlogs (*view_blog* for the /blog-posts endpoint and *marketing* for the /send-news-mails endpoint) and evaluated whether the frameworks enforce the limitation of processing to our predefined purposes.

RuleKeeper enforces purpose limitation with the help of OPA (see Listing 3 in Section A). For each application endpoint, the developers must specify a purpose in the application manifest. The authors provide five Rego-based tests that verify whether data operations comply with declared purposes; we confirmed all tests to execute correctly. We further modified the manifest of BreezeBlogs to simulate a purpose mismatch, accessing user interests under *marketing* instead of the declared *view blog*, and confirmed RuleKeeper blocked the operation, demonstrating effective enforcement.

Based on the developer-defined configuration, Fontus associates each sink with a purpose-recipient pair, i.e., who is processing and for what purpose. Incoming data is tainted with its allowed purpose and recipient pairs at the configured sources. This way, Fontus ensures that processing at the sink is only allowed if the pair matches. In our test, data tainted with *marketing/BreezeBlogs* was blocked at a sink configured for *view blog/BreezeBlogs*, confirming enforcement. This mechanism also controls third-party data sharing, since marking them as a sink would require a valid purpose.

GDPR-MFOTL implements purpose limitation within its MFOTL policy. Equation (2) from Section 3.4 shows the relevant rule. Unlike RuleKeeper or Fontus, for GDPR-MFOTL the developers do not have to define purposes centrally but embed them in the hosted application's code (see Listing 6 in Section A). We defined purposes for BreezeBlogs endpoints and registered 10 users who either consented to specific purposes or not. Only the consenting users' interests and mail addresses were returned when querying purpose-specific endpoints. Notably, only operations, not data, are mapped to purposes. Upon collection, the data lacks associated purpose metadata, making it impossible to determine the intended purpose. This limitation is evident in the *Collect* MFOTL predicate from the original paper, which takes only the data subject, data taint, and a special data flag as input. The authors rely on their browser extension (see Figure 8 in Section A) to obtain consent from the users. When users enter personal data via HTML forms, the extension intercepts the submission, extracts the data, and uniformly assigns consent for each toggled purpose to all fields. For example, if the *Marketing* purpose is toggled, every data item in the form receives consent for marketing, regardless of the data's actual use. While the extension technically allows per-field consent through a right-click context menu (Figure 9 in Section A), this feature is undocumented and does not resolve the core issue: data items are not purpose-bound and may be processed for any purpose once consent for that purpose is granted. Consequently, GDPR-MFOTL does not enforce purpose limitation.

Data Minimization. Data minimization requires that personal data is adequate, relevant, and limited to what is necessary for the purposes for which they are processed (Art. 5(1)(c)). This should take place not only while processing but also during the application's overall design. For example, a web application could implement an age check by having the user enter their age or through a yes/no question. Both would fulfil the purpose, but the latter collects less data (in case the web application does not need to verify the age). We cannot technically assess whether an application is designed with data minimization in mind, but we can evaluate whether this requirement is enforced within the application's processing. Neither Fontus nor GDPR-MFOTL claim to support data minimization, so we only evaluated RuleKeeper, which has a data minimization policy written in OPA's Rego language. For the operation to be performed, RuleKeeper receives the purpose of the operation and the corresponding maximum allowed data, specified by the developers in the manifest. It then verifies that the processed data does not exceed the maximum data. The naming *maximum data* is misleading because the maximum data may be more than the data necessary for the given processing purpose, instead of using the least amount of data required to fulfil the purpose. For BreezeBlogs, we defined the maximum data for *marketing* as the user's email. We tested data minimization with an operation that tries to fetch all the user's data from the database (*users.find()*) and an operation that only fetches the email from the database (*users.find({ username: requestedUsername }, {email: 1})*). Since email is the only data allowed to be processed for *marketing* purposes,

RuleKeeper blocked the database request trying to fetch all the user's data. Additionally, we tried to update the user's interests and found RuleKeeper blocking the database write. We conclude that RuleKeeper supports, implements, and enforces data minimization.

Storage Limitation. Storage limitation requires controllers and developers of a web application to only store data as long as it is necessary to fulfill the purpose the data is needed for (Art. 5(1)(e)). Fontus introduces the time period f in its metadata, which defines how long processing - and therefore storage - is valid for a given data item. However, the deletion of expired data is neither implemented nor enforced. Additionally, the deletion is not applicable for cases where it is not previously known how long it takes to fulfill the purpose. Setting f also requires the developers to know which data is needed for which amount of time. Neither of the other frameworks covers this aspect at all.

Integrity and Confidentiality. Integrity and confidentiality demand that personal data is processed with appropriate security to be safe against several threats like unauthorized processing, destruction, or accidental loss (Art. 5(1)(f)). We emphasize that the GDPR does not provide conclusive requirements for achieving integrity and confidentiality. It is a case-by-case weighing of whether a measure is sufficient. None of the three frameworks supports integrity, and only RuleKeeper claims to support confidentiality through a Rego policy provided by the authors. Each application operation is associated with a list of roles allowed to perform it. The user's role, stored in a database, is checked against this list, and if not permitted, the operation is blocked. Testing with BreezeBlogs verified this mechanism: the endpoint *POST /send-news-mails*, restricted to the *advertiser* role in the manifest, was accessible only to a user with that role, while a user with the *user* role was blocked. Additionally, the authors of RuleKeeper provided tests for their access control policy. For different roles, the tests try to access operations that should either be executable for the role or not, and verify that OPA [17] correctly blocks/allows executing the function for the given role. We conclude that RuleKeeper only implements and enforces confidentiality for use cases where access control is an adequate measure.

Lawfulness of Processing. The GDPR sets out multiple legal bases under which processing of personal data is allowed, including the data subject consenting to the processing or a contract making the processing necessary (Art. 6(1)). Despite allowing the developers to process personal data by specifying a legal basis, the frameworks do not make the basis transparent to the user to prove this. Additionally, developers may be unable to determine the legal basis for each purpose and need help from legal experts. However, if they can determine them correctly, the frameworks allow the developers to connect each purpose to a specific legal basis.

RuleKeeper has implemented a Rego policy that checks if the operation's purpose has either *consent* or *contract* as a legal basis (see Listing 5 in Section A). Misleadingly, it marks operations requiring consent even when the legal basis is *contract*, which does not necessitate user consent according to Art. 6(1)(b) as *contract* is a sufficient legal basis itself. To test consent-based processing, we set the purpose *view blog* in BreezeBlog's manifest to require consent. When requesting the endpoint */blog-posts*, data processing was blocked for users without consent, but allowed once consent was provided. This would also be the case for operations that provide data to server-side third parties, since RuleKeeper would only execute the third-party functionality if the user provided consent. RuleKeeper demands setting a legal basis manually; otherwise, all data processing is prohibited. For non-consent-based processing, we adjusted the purpose *view blog* to have the legal basis *legal obligation*. We found that data processing was permitted without user consent in this case. For further details on RuleKeeper's implementation of lawfulness of processing, see Listing 4 in Section A. In summary, RuleKeeper enforces data processing based on a legal basis in case developers can specify the correct basis for each purpose.

Fontus allows developers to add purposes resulting from, for example, the user entering into a contract with the website operator. This allows the developer to realize use cases where data processing is permitted for reasons beyond user consent, such as legitimate interests or a contract. This, too, relies on developers correctly setting up the framework. We tested consent-based processing, like for RuleKeeper, and found that Fontus also blocks processing without the user's consent.

To implement lawfulness of processing in GDPR-MFOTL, the authors introduced the formula *LegalGround(grd, ut, sp)* for developers to claim an arbitrary legal basis *grd* for a data item with the taint *ut*. We tested this by accessing a data item before and after declaring a legal basis. Processing was blocked initially but permitted afterward. For consent-based processing, we set up BreezeBlogs within GDPR-MFOTL, defined data processing based on consent, and found that accessing */blog-posts* was blocked for users without consent. Once consent was given, access was allowed. Thus, GDPR-MFOTL effectively supports and enforces lawfulness of processing.

Consent Handling. The GDPR sets out conditions that must be met for consent to be valid, including that it must be "freely given, specific, informed, and unambiguous" (Art. 4(11)). Additionally, systems must support mechanisms for consent collection, withdrawal, and the ability to demonstrate that valid consent has been obtained (Art. 7(1–4)) [5]. Consent is often obtained through a cookie banner, with the frameworks not being an exception. For cookie banners to be GDPR-compliant, several requirements have to be fulfilled [24]. Since assessing the compliance of a cookie banner is only partially possible from a technical standpoint, we focus on Art. 7(1-3). To obtain consent, RuleKeeper shows the user a cookie banner (Figure 6 in Section A) that explicitly asks for their consent when they access an endpoint that processes personal data and no consent has been provided yet. RuleKeeper does not implement a way for users to view their consent choices, but since the consent choices are stored and could be made available to the users, we state this requirement as supported. While the authors claim that users may withdraw their consent, this has not been implemented.

Like RuleKeeper, Fontus implements a consent dialogue (Figure 7 in Section A) that shows the recipient and a short description of the use for each purpose. Fontus does not allow users to view or change their past consent choices. However, since the user's consent choices are stored, we consider this as supported.

GDPR-MFOTL's dedicated dashboard in their privacy platform allows the data subject to explicitly set consent, to view consent

for each piece of data individually through its taint, and to revoke consent at any time. Besides that, GDPR-MFOTL also allows the user to view and change their consent choices through the browser extensions. We found all of the frameworks to be able to collect consent, but only GDPR-MFOTL to also enable the users to view, change, and withdraw their choices.

Special Data. The GDPR distinguishes between personal data and special personal data, which shall have a higher level of protection and should only be processed on an exceptional basis (Art. 9). RuleKeeper does not consider different types of personal data.

Fontus metadata introduces the required protection level *l*. With this parameter, developers can define a higher level of protection for special data. However, the GDPR does not explicitly require a higher level of protection for special data, but limits the legal bases under which the processing is allowed (Art. 9(2)(a–j)). Therefore, we consider handling special data to be supported, but neither implemented nor enforced.

The authors of GDPR-MFOTL introduce the concept of special data in their paper and include a special data flag in their *Collect* signature. However, we have not found any use of the flag in the source code, nor do processing signatures such as *Use* or *ShareWith* include the special data flag. Therefore, we consider handling special data to be supported, but neither implemented nor enforced.

Transparency. The controller must inform the data subject about the processing of their personal data and the current state of the execution of their rights as defined in Art. 12. RuleKeeper and Fontus do not support providing information to the data subject except when explicit consent is obtained. In addition, they do not enforce user rights and, therefore, do not inform data subjects about the current status of their rights' execution.

GDPR-MFOTL, on the other hand, has a user-accessible dashboard that shows the consent and processing of data at any time. In addition, when users want to execute their rights, it states when a request was received and when the right was executed. One weakness is that it only shows the taint of the collected data, not the actual values. Therefore, only GDPR-MFOTL supports, implements, and enforces information provision and action information.

Information to be Provided. The GDPR mandates 12 pieces of information that must be provided to the data subject when their personal data is obtained or processed (Art. 13).

RuleKeeper presents the user with a cookie banner each time personal data is processed or collected. It only shows the purpose of the processing, resulting in an overall score of 1/12.

Of the 12 pieces of information that should be provided to the data subject, Fontus only provides the purposes of the processing and the data recipients, resulting in a score of 2/12.

The browser extension of GDPR-MFOTL informs the data subject about the purposes of the processing, but nothing else. Therefore, it only provides 1/12 of the information to the data subject.

Data Subject Rights. The GDPR grants 6 data subject rights (Art. 15–18, 20–21), and in this subsection, we check whether the three frameworks support them. For each data subject right, the GDPR sets forth different modalities for how these rights must be exercised and fulfilled. As an example, the right of access defines which information must be provided to the data subject (Art. 15(1)(a–h)). There are also cases where the right of the data subject has to be enacted without the data subject needing to act, for example, in Art. 17(1)(b), where the withdrawal of consent must also result in deleting the data if there is no other valid legal basis to process it further. RuleKeeper does not support any of the data subject rights.

With the granular data tainting in Fontus, a data subject's personal data can be made available through database queries with the corresponding taints. However, the authors do not implement or enforce this in their framework for any data subject rights.

GDPR-MFOTL claims to implement all the demanded data subject rights through their privacy dashboard. Their approach combines the *right of access* and the *right to data portability* (Art. 15 and 20). For each tainted piece of data, the data subject may make an access request to receive their data. However, this is inconvenient for the data subject as the privacy dashboard only shows the taint, not the actual data value. The request gets automatically executed by GDPR-MFOTL, providing the data subject with their data, including a downloadable JSON file holding the data. However, the data is not combined in one JSON file; each data item has its own JSON file. Art. 15 requires multiple pieces of information to be provided to the data subject, for example, the purposes of processing and the recipients. Not all of the necessary information is provided to the data subject, and it is missing both the purpose and the recipient. We therefore consider the right of access to be supported. Since the data subject can export any of their data in a JSON file, arguably regarded as "commonly used and machine-readable" (Art. 20(1)), we consider the right to data portability as supported, implemented, and enforced. Inside the privacy dashboard, the data subject can arbitrarily change their personal data to any value. This allows the data subject to rectify false or incomplete data, resulting in the right to rectification being supported, implemented, and enforced.

For the right to erasure, GDPR-MFOTL allows the data subject to erase any data provided to the application. However, the right to erasure does not only cover the wishes of the data subject, but also, for example, if the personal data is no longer required for the processing purpose (Art. 17(1)(a)) or if the data subject withdrew their consent (Art. 17(1)(b)). While GDPR-MFOTL does not address these additional cases, the system primarily emphasizes the data subject's role in initiating erasure. This reflects one of the key individual empowerment goals of the GDPR, as emphasized in Recital 7 GDPR and Art. 1(2). We therefore consider this requirement as supported. For the right to restriction, GDPR-MFOTL enables the data subject to prevent processing of their data at any time. However, this also does not cover implicit cases, such as the data subject not wanting to have the data deleted, even if the purpose is fulfilled (Art. 18(1)(c)). As for the right to erasure, since the article primarily focuses on mechanisms that empower the data subject to restrict their data from processing, we consider this requirement to be supported. While GDPR-MFOTL claims to support the right to object, and we did find an endpoint in the codebase, but no possibility for the data subject to make use of this, we consider this as supported. We executed each right through GDPR-MFOTL's privacy dashboard to test their execution and found them, besides the above points, to execute correctly.

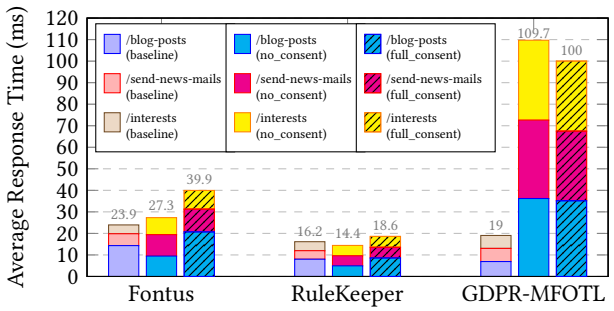


Figure 4: Mean Response Time for a Single User per Endpoint.

Records of Processing Activities. The GDPR mandates that the controller maintains a record of processing activities (Art. 30). However, this record does not have to be accessible to the end user, but shall be made "available to the supervisory authority on request" (Art. 30). Of the three frameworks, only Fontus and GDPR-MFOTL claim to support this requirement. The record of processing activities should contain a total of 7 pieces of information (Art. 30).

Fontus partially covers this requirement with its configuration file. Its configuration contains the purposes and recipients of the data processed by the application. Therefore, Fontus provides 2/7 of the pieces of information. GDPR-MFOTL also claims to support this requirement, but does not specify how this is implemented. We assume that the authors consider the operation log (see Listing 2 in Section 3.4) as the record here, which shows the purpose of processing, 1/7 of the pieces of information.

5.2 Runtime Overhead

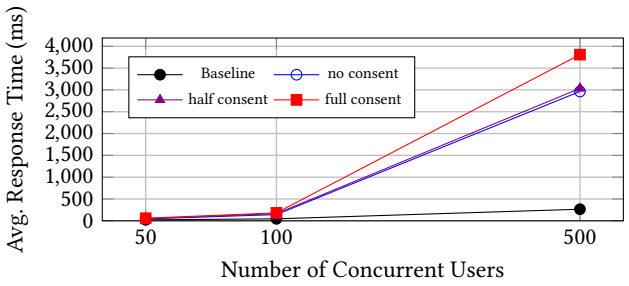
In this section, we evaluate and compare the runtime overhead introduced by the three frameworks. As described in Section 4.1, we integrated BreezeBlogs into each GDPR enforcement framework. We then measured the response times for each combination of BreezeBlogs and GDPR enforcement framework, as well as a baseline version of BreezeBlogs without GDPR enforcement, for 13 minutes using Locust.

In the single-user benchmark, we measured the response times of the three endpoints for a single user for each framework. We did this for the cases where the user has given consent for processing their data, where the user has not given consent, and without any GDPR enforcement in place to measure the impact of GDPR enforcement based on the user's consent choices. Figure 4 shows the average response times for the three endpoints stacked on top of each other. The response times were measured in milliseconds, and the overhead was calculated as the sum of the endpoint response times of the framework and the standalone application. The measurements show that RuleKeeper introduced the lowest overhead for the single-user scenario, with even a negative overhead for the benchmark without the user's consent.

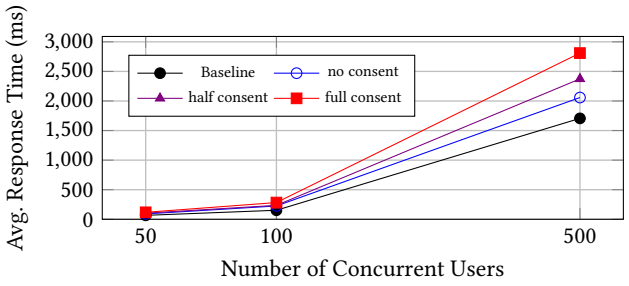
Zooming in on the individual endpoints for a single user who has not given any consent, we see that the overhead for the */blog-posts* endpoint was negative for both RuleKeeper (-38.3%) and Fontus (-33.92%). This is the case because, without consent, the blog posts were not fetched and processed, so the response time was lower than for BreezeBlogs running standalone. For GDPR-MFOTL, however, the overhead for this endpoint was high due to the consent mechanism. Not fetching the blog posts was insufficient to compensate for the overhead introduced by the framework. The overall overhead for GDPR-MFOTL was higher than for the other two frameworks, with a maximum overhead of 517.89% for the */interests* endpoint. While the other frameworks perform the enforcement inside the backend, GDPR-MFOTL has to connect to its enforcement backend via HTTP, which could lead to a higher latency. For all three frameworks, the overhead of the */blog-posts* endpoint was lower than that of the */interests* endpoint. The */blog-posts* endpoint had to do more computation for BreezeBlogs running standalone, because when running within the frameworks, the blog posts were not fetched and processed due to a lack of user consent. In this case, GDPR-MFOTL was the only framework with a higher absolute response time for the */interests* endpoint than for the */blog-posts* endpoint. Our single-user benchmark showed significant differences between the three frameworks in terms of response times. Besides using different programming languages and libraries, this may be due to the different designs of the frameworks' enforcement mechanisms. Fontus leverages pre-function hooks to check if processing is allowed/denied. RuleKeeper and GDPR-MFOTL on the other hand, both leverage different services (see Figure 1 in Section 3.2 and Figure 3 in Section 3.4) which are responsible for the enforcement. RuleKeeper uses web sockets for the communication between its services, while GDPR-MFOTL relies on HTTP requests. Additionally, GDPR-MFOTL stores all traces in a database. To know if there was an event from a data subject who provided consent, it has to search the whole trace database to be sure that this event ever occurred. However, we did not measure the components individually due to their close interaction inside the frameworks.

We also measured the response times for the multi-user benchmarks for the three endpoints for each framework. We did this for the case where users have given consent to process their data, the case where users have not given consent, and the case where half of the users have given consent. Due to the different execution times, the total collected response times differ between the frameworks. For example, this leads to 153 986 measurements for RuleKeeper running with 100 concurrent users. While evaluating GDPR-MFOTL with multiple users, we encountered race conditions that crash the application. The authors of GDPR-MFOTL acknowledged this issue but did not provide a fix in time for the evaluation. Figure 5 shows the average response times for (a) Fontus and (b) RuleKeeper for the multi-user benchmarks for the three different consent scenarios. Our measurements show a higher discrepancy between the application's response times with the frameworks under the different consent scenarios than when the application runs standalone. We found differences, for 500 users, ranging from 2698.71 ms to 3543.08 ms for Fontus, while for RuleKeeper, the differences range from 352.22 ms to 1103.89 ms. We also see from the measurements that for Fontus, while the response times for the *no_consent* and *half_consent* scenarios are close together, there is a gap between these two and the *full_consent* scenario.

For RuleKeeper, the gaps between the response times of the three scenarios are more evenly distributed. Table 2 shows the percentage overheads for Fontus and RuleKeeper for the different consent scenarios and user counts.



(a) Average Response Times per Consent Scenario for Fontus.



(b) Average Response Times per Consent Scenario for RuleKeeper.

Figure 5: Response Times for Fontus and RuleKeeper (Baseline is BreezeBlogs without GDPR Enforcement Tool).

While both frameworks added overhead for the different scenarios, Fontus introduced more overhead than RuleKeeper. Except for the *no consent* scenario, Fontus at least doubled the computation time for the application, while RuleKeeper added at most an overhead of 85.6%. In addition, the overhead for Fontus increased with the number of users, while for RuleKeeper it decreased.

Overall, our multi-user benchmarks shows that RuleKeeper introduced less overhead than Fontus for the different scenarios and user counts. In addition, the overhead for RuleKeeper decreased with the number of users, while the overhead for Fontus increased.

Table 2: Runtime Overheads for Fontus and RuleKeeper per Consent Scenario.

#Users	Overhead in ms	
	Fontus	*RuleKeeper*
	full consent	
50	36.4 (165.3%)	51.3 (77.2%)
100	137.2 (304.6%)	130.5 (85.6%)
500	3543.1 (1334.8%)	1103.8 (64.7%)
	half consent	
50	29.4 (133.5%)	36.6 (55%)
100	118.3 (262.6%)	83.2 (54.6%)
500	2768.9 (1043.2%)	667.3 (39.1%)
	no consent	
50	21.8 (98.9%)	24.3 (36.5%)
100	97.5 (216.3%)	72.8 (47.8%)
500	2698.7 (1016.7%)	352.2 (20.7%)

One reason for this is how RuleKeeper implements the consent mechanism (for more details on this, we refer to Section B.1).

5.3 Storage Overhead

We also evaluated the storage overhead introduced by the frameworks. To do this, we measured the amount of space required by BreezeBlogs standalone and when integrated with each framework.

RuleKeeper. The storage layout of RuleKeeper relies on MongoDB collections. We implemented BreezeBlogs with three collections, namely *advertisers*, *blogposts*, and *users*. The *advertisers* collection holds users allowed to access marketing-related endpoints (/send-news-mails), while the *users* collection holds all user-specific data (username, password, email, interests). Lastly, the *blogposts* collection contains the blogpost entries. RuleKeeper adds three collections: *principals*, *entities*, and *consents*. With the *principals* and *entities* collections, RuleKeeper connects different types of functionality, like access control and distinguishing between data subjects and the controller. RuleKeeper adds those three collections regardless of the hosted web application's implementation. We measured the total size of the collections used by BreezeBlogs and those introduced by RuleKeeper, with the results shown in Table 3. Doubling from 50 to 100 users increased the overhead by a factor of about 1.9, while going from 100 to 500 users only increased the overhead by a factor of about 3.5.

Fontus. The database structure of BreezeBlogs is the same as described previously for RuleKeeper, but implemented as MySQL tables instead of MongoDB collections. Fontus does not add new tables but alters the existing ones by adding a taint field for every existing database field. We measured the overhead by registering the users in Fontus to correctly set up the taint values, leading to the results shown in Table 3. While we see that Fontus added more overall overhead than RuleKeeper, it was also the case that after a steep increase of overhead (50 to 100 users with a factor of 33), the overhead decelerated (100 to 500 user with a factor of about 1.2). While this may be due to MySQL optimizations, the growth did not scale with the number of users registered within the application.

GDPR-MFOTL. While GDPR-MFOTL also follows a data tainting approach, it alters the storage backend differently than Fontus by combining SQLite and QuestDB. The SQLite table holds BreezeBlogs data (with the same tables as for the other frameworks) and adds additional tables to store taint values. This adds 27 tables to the BreezeBlogs SQLite file. Due to the design of GDPR-MFOTL, it also has a table for users to authenticate to GDPR-MFOTL, rather than directly to BreezeBlogs, to access the privacy dashboard. Besides the SQLite database, GDPR-MFOTL also stores the traces in a QuestDB database to track specific events' timings. The results of our storage evaluation are shown in Table 3. We see that GDPR-MFOTL had a higher storage overhead regarding BreezeBlogs. However, by default, QuestDB pre-allocated a high amount of storage (2.4 GB), resulting in distorted results. Additionally, the size of the QuestDB database did not vary between 50 and 500 users. For the storage overhead results without QuestDB, we refer to Section B.3.

Table 3: Storage Overhead for all Frameworks.

#Users	RuleKeeper				Fontus				GDPR-MFOTL			
	W (kB)	*W/o (kB)*	*O (kB)*	*O (%)*	*W (kB)*	*W/o (kB)*	*O (kB)*	*O (%)*	*W (kB)*	*W/o (kB)*	*O (kB)*	*O (%)*
50	122	109	12	11.3	889	610	279	45.6	2524 k	164	2523 k	1 540 153.2
100	140	116	25	21.3	9933	610	9322	1527.5	2524 k	164	2523 k	1 540 180.7
500	290	166	124	74.4	12 030	643	11 387	1770.7	2524 k	229	2524 k	1 100 255.8

W: with framework, W/o: without framework, O: overhead

6 Discussion

In this section, we discuss the results of our evaluation and the implications of our findings, including the GDPR coverage of the frameworks and their runtime and storage overhead. Additionally, we provide insights into the required efforts to set them up.

GDPR Coverage. Our evaluation shows that Fontus, RuleKeeper, and GDPR-MFOTL support GDPR compliance by intercepting data processing functions and tracking personal data. However, while they all address several of the GDPR's requirements, we found that some of the investigated GDPR articles lack support or have limited implementation. For example, RuleKeeper does not support data subject rights, while Fontus' introduced metadata tuple can cover most of the evaluated GDPR requirements, but does not implement them. While we found that GDPR-MFOTL offers the most extensive coverage, both Fontus and GDPR-MFOTL utilize data tainting to enforce GDPR requirements, suggesting that Fontus could be developed further to support the same requirements as GDPR-MFOTL. In addition, both Fontus and GDPR-MFOTL could be extended to process and share data across different applications and services while keeping the taints, and therefore the metadata and consent choices, intact. This would be a significant advantage over RuleKeeper, which is limited in tracking data flows throughout one application. We have sent a draft of this work to the authors of all three frameworks, allowing them to provide feedback and to perform a sanity check on our findings and the description of their systems. Their feedback helped validate the technical details we reported. Additionally, the authors of GDPR-MFOTL acknowledged certain limitations in their framework, specifically regarding support for purpose limitation and the inability to handle multiple users concurrently.

Recommendations: For some of the GDPR requirements examined, the implementation and sometimes the interpretation of the framework authors differ. Future work should focus on (1) enhancing the frameworks, e.g., by adding support for missing requirements, and (2) fostering more cross-disciplinary collaboration with legal experts to promote shared interpretations and reduce ambiguity in future technical implementations.

Runtime & Storage Overhead. Our evaluation showed that the frameworks added significant runtime overhead to our test application, BreezeBlogs. While this overhead is unavoidable due to the frameworks' nature as additional components, it is important to consider when deploying them in a production environment. RuleKeeper supports the fewest GDPR requirements and introduced the least overhead, even speeding up the application in some cases. Due to the granular data-tainting approach, Fontus did not scale well with the number of users. For GDPR-MFOTL, we found that it introduces significant overheads, and, due to race conditions, we were unable to evaluate it in a multi-user context, which is crucial for a modern web application. However, it was also the most comprehensive in terms of GDPR coverage. We found significant differences in the introduced overheads, ranging from about 11% to 1 540 180%. However, comparing them is challenging since each framework relies on a different (but state-of-the-art) database system with varying configurations for pre-allocation, programming languages, or other system-introduced overheads. We can tell that there was a noticeable difference between RuleKeeper and both tainting approaches, with the latter introducing a higher overhead in every case.

Recommendations: Since RuleKeeper introduces the least runtime overhead, we recommend relying on existing technology as much as possible, since mature software may already be optimized for runtime and storage overhead. We do not want to argue against certain technologies like tainting or relying on specific database systems. The overhead from GDPR-MFOTL mostly comes from their QuestDB backend. However, the database system is suitable for the use case and should not be changed solely based on a higher storage overhead.

Development Overhead. Since we deployed the frameworks ourselves, we also want to provide some insights into the required effort to set them up. All three frameworks required us to contact their respective authors, primarily because documentation was missing and the published code did not run without errors. This is a significant barrier for developers who want to use any of the frameworks for their applications. RuleKeeper's deny-by-default approach forces developers to specify every data processing functionality inside the GDPR manifest; otherwise, the functionality is not executed. While this can lead to frustration when setting up RuleKeeper, it is ultimately the least error-prone approach, as it ensures that no data processing functionality is overlooked and executed by mistake. On the other hand, setting up Fontus and GDPR-MFOTL is more error-prone, as they rely on the developers to correctly tag all data and write the functionality to block or allow processing themselves. This can lead to oversights, such as forgetting to specify sinks or sources, resulting in incorrect data processing or even GDPR violations. We believe this is a significant drawback of Fontus and GDPR-MFOTL, since they do not guarantee that data is processed only in a GDPR-compliant way. Besides the effort required to set up the frameworks, the overhead is also introduced by the necessity of human involvement. To achieve purpose limitation, developers need to specify the correct purposes, including the required data, and, in combination with the lawfulness of

processing, the legal basis on which processing is allowed. For this, the frameworks rely on strings like *contract* or *legal_obligations*, which may not be sufficient to present the basis. This is also difficult in the context of fast-moving development, pushing out features as soon as possible to be ahead of the competition. As past research has shown (e.g., 4, 12, 21, 23, 27), operators and developers struggle to comply with the GDPR, making these questions a pressing issue. Li et al. [20] found that developers see privacy compliance as costly (i.e., having to adapt their applications to new regulations) while providing little perceived benefits. Horstmann et al. [13] showed that even with access to a privacy expert, developers struggle to implement compliant software solutions.
Recommendations: Following our experience, we recommend that researchers focus more on providing information on how to set up their frameworks. This includes but is not limited to (1) providing a minimal working example in a standardized format (e.g., docker-compose) and (2) providing a step-by-step instruction for integrating the framework into a web application.

Threats to Validity. Similar to the three frameworks, we, too, worked with a legal expert to derive the list of requirements (Contribution (i)). However, interpreting the law is not trivial and ultimately the judiciary's responsibility; hence, our list might not have captured all the nuances and implicit requirements of the GDPR. Additionally, our application, BreezeBlogs, is simple and has basic functionality. While this is advantageous because it allows easy testing of different aspects due to its low complexity, it is unsuitable for covering more advanced processing, including implicit data flows and data combination. Moreover, the technologies used also play a crucial role: Both JavaScript (RuleKeeper) and Java (Fontus) have optimizing runtimes, utilizing a Just-in-Time compiler to improve performance, while PythonTTC does not. Consequently, some of the additional overhead of GDPR-MFOTL might be due to the choice of the PythonTTC programming language. The same argument holds for choice of databases: Both MongoDB and MySQL, used by RuleKeeper and Fontus respectively, are built for high concurrent loads and have optimizing query planners, while SQLite, used by GDPR-MFOTL, might negatively impact GDPR-MFOTL's relative performance. Moreover, none of the frameworks evaluated are applicable in a scenario involving client-side third parties. Finally, as mentioned in Section 3.1, our literature review led to papers published in top-tier security and privacy venues. However, focusing on top-tier security and privacy venues may have excluded relevant GDPR enforcement papers published in other research communities, such as software engineering.

7 Related Work

We now provide an overview of the current field of research on privacy in the context of web applications, specifically in the scope of the GDPR. To the best of our knowledge, there is no previous work comparing GDPR enforcement frameworks. We focus on two distinct approaches addressing different aspects of the GDPR. First, we examine the concept of policy-based privacy management, where machine-readable privacy policies, formal rules defining how personal data should be handled, are used to control the behavior of a web application. Second, we discuss how data subject rights may be enforced since they are a central aspect of the GDPR.

Policy-based Privacy Management. Giffin et al. [9] proposed *Hails*, a framework for policy-based data flows through multiple web applications. Within *Hails*, developers define policies that restrict the data flow between different web applications and enforce these policies at runtime. This enables untrusted third-party applications to access data from a trusted application, ensuring that only the data allowed by the policy is shared. However, *Hails* does not provide a way to enforce privacy policies in the applications themselves, only on the shared data between them. Consequently, it does not offer a way to enforce the GDPR requirements on the applications. Another approach was taken by Wang et al. [29]. They proposed *Riverbed*, a system that allows users to define their own privacy policies for web applications, stating how their data can be shared or stored by the application. *Riverbed* uses a proxy on the client side and a trusted execution environment on the server side to enforce these policies. While this covers the aspect of data minimization and provides the data subject with control over their data, it does not enforce different data subject rights demanded by the GDPR.

Enforcing Data Subject Rights. Truong et al. [28] proposed an approach, based on a blockchain, to enforce the right of access, restriction, data portability, and erasure. They also claimed to support the *Right to be informed*, which is not a specified data subject right in the GDPR but appears to be a reference to Art. 13, demanding that the data subject is informed about the processing of their data. Furthermore, they introduced a central platform between the blockchain, the data subject, and the data controller. The platform, for example, receives a consent request from the data controller, which is then forwarded to the data subject, whose decision is then stored on the blockchain. This category also includes the three GDPR compliance enforcement frameworks discussed in this work: RuleKeeper [7], GDPR-MFOTL [15], and Fontus [19].

8 Conclusion

In this work, we evaluated three recently published frameworks that assist developers with GDPR compliance. We compared the frameworks based on their GDPR coverage and both the computational and development overhead. Our primary takeaway is that each framework presents a unique balance of strengths and weaknesses. While introducing the least computational overhead, RuleKeeper supports the fewest GDPR requirements, including none of the data subject rights. In contrast, Fontus and GDPR-MFOTL support more of the requirements (despite needing more engineering to enforce them), at the cost of increased overhead. With its data-tainting approach, Fontus shows promise due to its comprehensiveness and flexibility. However, it suffers from scalability issues with increasing numbers of users. GDPR-MFOTL covers the most GDPR aspects but introduces significant computational and development overhead, even requiring developers to learn a new language, PythonTTC. While highly promising, we conclude that none of the analyzed frameworks are deployable in their current form. This offers interesting opportunities for future research, both in terms of optimizing such enforcement approaches to improve scalability and exploring how to simplify the integration process and minimize the learning curve for developers. Both factors are crucial for broader adoption, ensuring robust GDPR compliance while maintaining performance and ease of development.

Acknowledgments

We are thankful for the valuable feedback and suggestions of our anonymous shepherd and reviewers. We also thank Dr. Maxi Nebel from the University of Kassel for helping us interpret the GDPR and answering all our law-related questions. We gratefully acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972.

References

[1] Tiago Bianchi. 2024. Advertising Revenue of Google from 2001 to 2024. https://www.statista.com/statistics/266249/advertising-revenue-of-google/ Accessed on 2024-01-29.

[2] European Commission. 2023. Exchange of Electronic Health Records across the EU | Shaping Europe's Digital Future. https://digital-strategy.ec.europa.eu/en/policies/electronic-health-records Accessed on 2024-01-29.

[3] European Commission. 2024. European Digital Identity - European Commission. https://commission.europa.eu/strategy-and-policy/priorities-2019-2024/europe-fit-digital-age/european-digital-identity_en Accessed on 2024-01-29.

[4] 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. *Proceedings on Privacy Enhancing Technologies* 2022, 95–113. https://doi.org/10.2478/popets-2022-0037

[5] European Data Protection Board. 2020. Guidelines 05/2020 on consent under Regulation 2016/679. https://www.edpb.europa.eu/sites/default/files/files/file1/edpb_guidelines_202005_consent_en.pdf

[6] European Parliament and Council of the European Union. 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). OJ L 119, 4.5.2016, pp. 1–88. https://data.europa.eu/eli/reg/2016/679/oj.

[7] Mafalda Ferreira, Tiago Brito, José Fragoso Santos, and Nuno Santos. 2023. RuleKeeper: GDPR-Aware Personal Data Compliance for Web Frameworks. In *Proceesings of the 2023 IEEE Symposium on Security and Privacy (SP '23)*. IEEE Computer Society, USA, 2817–2834. https://doi.org/10.1109/SP46215.2023.10179395

[8] Mafalda Ferreira, Tiago Brito, José Fragoso Santos, and Nuno Santos. 2024. Rulekeeper GitHub Repository. https://github.com/rulekeeper/rulekeeper Accessed on 2024-05-22.

[9] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. 2012. Hails: Protecting Data Privacy in Untrusted Web Applications. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 47–60. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/giffin

[10] Johannes Gille, Sebastian Meineck, and Ingo Dachwitz. 2023. EU country comparison: How data brokers are screening us. https://netzpolitik.org/2023/eu-country-comparison-how-data-brokers-are-screening-us/ Accessed on 2024-01-29.

[11] Daniel Hamlin and Paul E. Peterson. 2022. Homeschooling Skyrocketed during the Pandemic, but What Does the Future Hold? https://www.educationnext.org/homeschooling-skyrocketed-during-pandemic-what-does-future-hold-online-neighborhood-pods-cooperatives/ Accessed on 2024-01-29.

[12] Stefan Albert Horstmann, Samuel Dominiks, Marco Gutfleisch, Mindy Tran, Yasemin Acar, Veelasha Moonsamy, and Alena Naiakshina. 2024. "Those things are written by lawyers, and programmers are reading that." Mapping the Communication Gap Between Software Developers and Privacy Experts. *Proceedings on Privacy Enhancing Technologies* 2024, 151–170. https://doi.org/10.56553/popets-2024-0010

[13] Stefan Albert Horstmann, Sandy Hong, David Klein, Raphael Serafini, Martin Degeling, Martin Johns, Veelasha Moonsamy, and Alena Naiakshina. 2025. "Sorry for bugging you so much." Exploring Developers' Behavior Towards Privacy-Compliant Implementation . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1159–1177. https://doi.org/10.1109/SP61157.2025.00146

[14] François Hublet, David Basin, and Srđan Krstić. 2022. Real-Time Policy Enforcement with Metric First-Order Temporal Logic. In *Computer Security – ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part II* (Copenhagen, Denmark). Springer-Verlag, Berlin, Heidelberg, 211–232. https://doi.org/10.1007/978-3-031-17146-8_11

[15] François Hublet, David Basin, and Srđan Krstić. 2024. Enforcing the GDPR. In *Computer Security – ESORICS 2023: 28th European Symposium on Research in Computer Security, The Hague, The Netherlands, September 25–29, 2023, Proceedings, Part II* (The Hague, The Netherlands). Springer-Verlag, Berlin, Heidelberg, 400–422. https://doi.org/10.1007/978-3-031-51476-0_20

[16] François Hublet, David Basin, and Srđan Krstić. 2024. User-Controlled Privacy: Taint, Track, and Control. *Proceedings on Privacy Enhancing Technologies* 2024, 1, 597 – 616. https://doi.org/10.3929/ethz-b-000641987 24th Privacy Enhancing Technologies Symposium (PETS 2024); Conference Location: Bristol, UK; Conference Date: July 15-20, 2024.

[17] Styra Inc. 2024. Open Policy Agent. https://www.openpolicyagent.org/ Accessed: 2024-05-22.

[18] Joakim Hamrén Jonatan Heyman, Carl Byström and Hugo Heyman. 2024. Locust.io. https://locust.io/ Accessed: 2024-11-29.

[19] 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* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 3343–3357. https://doi.org/10.1145/3576915.3616604

[20] Tianshi Li, Elizabeth Louie, Laura Dabbish, and Jason I. Hong. 2021. How Developers Talk About Personal Data and What It Means for User Privacy: A Case Study of a Developer Forum on Reddit. *Proc. ACM Hum.-Comput. Interact.* 4, CSCW3, Article 220, 28 pages. https://doi.org/10.1145/3432919

[21] 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 *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*. USENIX Association, Santa Clara, CA, 371–385. https://www.usenix.org/conference/soups2019/presentation/dimartino

[22] Matthew Rosenberg, Nicholas Confessore, and Carole Cadwalladr. 2018. How Trump Consultants Exploited the Facebook Data of Millions. https://www.nytimes.com/2018/03/17/us/politics/cambridge-analytica-trump-campaign.html Accessed on 2024-07-27.

[23] Marlene Saemann, Daniel Theis, Tobias Urban, and Martin Degeling. 2022. Investigating GDPR Fines in the Light of Data Flows. *Proceedings on Privacy Enhancing Technologies* 2022, 314–331. https://doi.org/10.56553/popets-2022-0111

[24] Cristiana Santos, Nataliia Bielova, and Célestin Matte. 2020. Are cookie banners indeed compliant with the law? : Deciphering EU legal requirements on consent and technical means to verify compliance of cookie banners. *Technology and Regulation* 2020 (Dec. 2020), 91–135. https://doi.org/10.71265/g317tv72

[25] 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 *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*. IEEE Computer Society, USA, 317–331. https://doi.org/10.1109/SP.2010.26

[26] Frances Stead Sellers. 2015. Cruz Campaign Paid $750,000 to 'Psychographic Profiling' Company. https://www.washingtonpost.com/politics/cruz-campaign-paid-750000-to-psychographic-profiling-company/2015/10/19/6c83e508-743f-11e5-9cbb-790369643cf9_story.html Accessed on 2024-07-27.

[27] 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. *Proceedings on Privacy Enhancing Technologies* 2021, 351–372. https://doi.org/10.2478/popets-2021-0051

[28] Nguyen Binh Truong, Kai Sun, Gyu Myoung Lee, and Yike Guo. 2020. GDPR-Compliant Personal Data Management: A Blockchain-Based Solution. *IEEE Transactions on Information Forensics and Security* 15 (2020), 1746–1761. https://doi.org/10.1109/TIFS.2019.2948287

[29] Frank Wang, Ronny Ko, and James Mickens. 2019. Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 615–630. https://www.usenix.org/conference/nsdi19/presentation/wang-frank

A Framework Details

In this section, we present additional details of the frameworks.

A.1 RuleKeeper

Figure 6 shows the consent banner generated by RuleKeeper for an endpoint that wants to process the personal data "user interests" for the purpose "view interests". Although the banner informs users about the intended data processing, it raises concerns about GDPR compliance. Specifically, the banner does not appear to offer a clear option to deny consent, and the use of the label "Submit" may not

provide sufficiently unambiguous communication regarding the user's choice.

Figure 6: RuleKeeper Consent Banner.

Listing 3 shows the purpose limitation policy of RuleKeeper. For each operation called, RuleKeeper checks whether the purpose of the processed personal data matches the purpose of the operation.

```
package rulekeeper

#default allowPurposeLimitation = false

# We want to allow the query if the purposes of the data being processed include the purpose of the operation
allowPurposeLimitation(personalData) {
        # Get operation
        operation := getOperation(input.operation)
        # Check purpose of the invoked operation
        purpose := getOperationPurpose(operation)
        count(purpose) >= 1
        # Get collected purposes of the personal data (union)
        data_purposes := {p | p = data.data_purposes[_]; p.data == personalData[_]}
        # Check if processed all data purposes contain the purpose of the operation
        res := {x | x = data_purposes[_]; x.purposes[_] == purpose}
        count(res) == count(personalData)
}
```

Listing 3: RuleKeeper Purpose Limitation Rego Policy.

Listing 4 shows the Rego policy used by Rulekeeper to enforce lawfulness of processing. The Rego policy allows processing under three circumstances, which are all checked in order of appearance in the policy: (1) the operation does not require consent and there is a legal base defined for the operation in the applications manifest, (2) the principal is a data subject for which the policy checks whether the data subject has given consent for the data processing purpose or not, (3) the principal is not a data subject and RuleKeeper checks if all data subjects of the data involved have provided consent.

Listing 5 shows the Rego function that determines whether an operation requires consent or not. It first gets the operation and the purpose and afterwards determines the legal basis for which processing is allowed. If the base equals "consent" or "contract", RuleKeeper marks the operation as requiring consent, which is misleading as these are two different legal bases and we do not know why RuleKeeper is implemented this way.

```
package rulekeeper

default allowLawfulnessOfProcessing = false

# Allow if the lawfulness base is valid, and if it is, if the data subject gave consent, if needed
# exists p in purposes(o) and requires-consent(p) then, for all d in dataset(o), granted-consent(owner(d), d, p)

# Allow if the lawfulness base is valid and does not require consent verification
allowLawfulnessOfProcessing {
        not operationRequiresConsent
}
# Allow if the lawfulness base is valid and the data subject consented
allowLawfulnessOfProcessing {
        # If the principal is a data subject - check its consent
        principalIsDataSubject
        # Get operation
        operation := getOperation(input.operation)
        # Check purpose of the invoked operation
        purpose := getOperationPurpose(operation)
        # Check consent of the entity associated with the principal
        subjectConsent := getConsent(input.principal)
        # Check if subject has a valid lawfulness base for the purpose of the operation
        purpose == subjectConsent[_]
}

# Allow if the lawfulness base is valid and all the data subject consented
allowLawfulnessOfProcessing {
        # If the principal is not the data subject - check consent of all subjects involved
        principalIsControllerProcessor
        # Get operation
        operation := getOperation(input.operation)
        # Check purpose of the invoked operation
        purpose := getOperationPurpose(operation)
        # Get consent of the subjects
        subjectConsents := getSubjectsConsent
        # Check if all subjects have a valid lawfulness base for the purposes of the operation
        numberConsented := [consent | consent = subjectConsents[_]; purpose == consent[_]]
        # Check if all subjects have a valid lawfulness base for the purposes of the operation
        count(numberConsented) == count(input.subjects.list)
}
```

Listing 4: RuleKeeper Lawfulness of Processing Rego Policy.

```
# Check if operation requires consent
operationRequiresConsent {
    # Get operation purpose
    operation := getOperation(input.operation)
    purpose := getOperationPurpose(operation)
    # Get purpose lawfulness base
    base := getPurposeLawfulnessBase(purpose)
    base == ["consent", "contract"][_]
}
```

Listing 5: RuleKeeper Operation Requires Consent Rego Policy.

A.2 Fontus

Figure 7 displays the consent dialog for Fontus that asks the data subject to provide consent for each of the consent-based purposes.

While the dialog allows users to individually consent to different purposes, it remains open to question whether the way consent is collected fully satisfies GDPR standards.

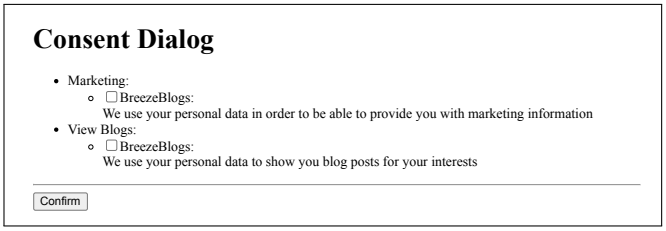


Figure 7: Fontus Consent Dialog.

A.3 GDPR-MFOTL

Figure 8 shows the browser extension used in GDPR-MFOTL in order for the data subject to provide/change their data choices.

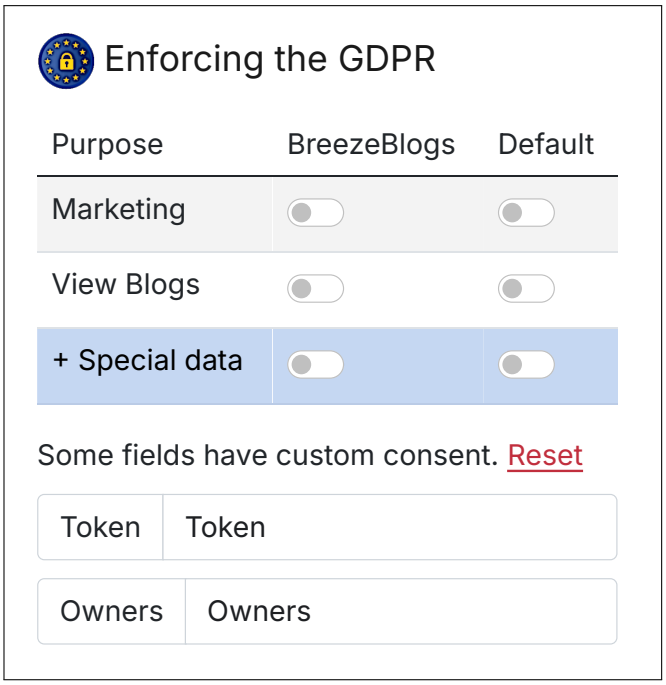


Figure 8: GDPR-MFOTL Browser Extension.

In order to only process data for a specific purpose a legal basis applies to, GDPR-MFOTL implements a *check_all* function which gets the data to be processed and filters out the data for which processing is not allowed, shown in Listing 6.

GDPR-MFOTL provides the user with the ability to set individual consent for the form keys, which refer to input fields in a web form (e.g., name, email, address). However, this feature is undocumented and requires the user to right-click on the field for an additional context menu as shown in Figure 9.

```
def filter_check_marketing(messages):  # sp
    messages2 = []
    checks = mockapp.check_all("marketing", messages)
    i = 0
    while i < len(messages):
        if checks[i]:
            append(messages2, messages[i])
        i += 1
    return messages2
```

Listing 6: GDPR-MFOTL Purpose Filter.

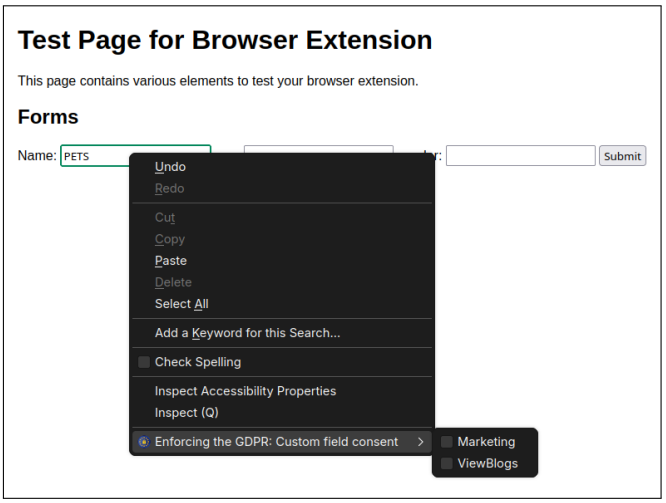


Figure 9: GDPR-MFOTL Browser Extension Context Menu.

B Additional Experiments

In this section we present additional experiments that we have carried out.

B.1 RuleKeeper Measurements per Endpoint

To investigate further on the endpoints response time overheads, we looked at */blog-posts* and */send-news-mails* individually for the different scenarios and user counts, with the results shown in Table 4.

The table shows the response time overhead for each endpoint, each consent scenario, and each number of concurrent users. The overhead is calculated between the average response time of the endpoint and the response time without RuleKeeper. In addition, the last three rows show the average content size sent by BreezeBlogs running standalone. For the */blog-posts* endpoint and the full consent scenario, the same number of bytes is returned as from BreezeBlogs running standalone (except for the 500 users scenario). For the other scenarios, the lower the number of users with consent, the lower the amount of data returned, as the blog posts are not fetched from the database and processed. Here the difference between the two endpoints is noticeable. For the */send-news-mails* endpoint, if and only if each individual user has consented to the processing, the amount of data returned will be the same as the amount of data returned by BreezeBlogs running standalone. This is due to the way RuleKeeper implements the consent mechanism

Table 4: Endpoint Measurements for RuleKeeper.

Scenario	#Users	/blog-posts		/send-news-mails	
		RTO (%)	*ACS (byte)*	*RTO (%)*	*ACS (byte)*
full consent	50	82.91	103 115	73.92	1229
	100	86.48	103 115	88.54	2479
	500	45.94	102 884	62.35	13 279
half consent	50	35.24	48 904	64.73	41
	100	32.53	48 268	68.13	41
	500	−4.79	46 642	55.75	41
no consent	50	−7.44	41	61.79	41
	100	−11.46	41	90.00	41
	500	−23.50	41	25.92	41
without framework	50		103 115		1229
	100		103 115		2479
	500		103 115		13 279

RTO: Response Time Overhead, ACS: Average Content Size

when data from multiple data subjects is fetched from the database at once. Listing 7 shows the code snippet from the enforcement policy where RuleKeeper checks whether the processing of the data is lawful. The code snippet shows that the processing is only allowed if all the data subjects have consented to the processing. Since the endpoint implementation fetches all user email addresses at once, the processing is only allowed if all users have given consent.

```
# Check if all subjects have a valid lawfulness base for the purposes
↪  of the operation
numberConsented := [consent | consent = subjectConsents[_]; purpose
↪  == consent[_]]
# Check if all subjects have a valid lawfulness base for the purposes
↪  of the operation
count(numberConsented) == count(input.subjects.list)
```

Listing 7: RuleKeeper Verify Lawfulness of Processing for Multiple Data Subjects [8].

B.2 Registration Phase

In addition to the runtime benchmarks, we also measured the time taken to set up a new user for the application. To do this, we first identified which functions/endpoints to call and in what specific order to register a new user, both for the standalone application and then for the application hosted within each framework. We then repeatedly measured the time it took to register new users for each framework and the standalone application for a total of 13 minutes. The first 3 minutes are the warm-up phase and are not included in the results. For RuleKeeper, we did not find a way to add a new user to the application. However, we found the functionality to insert a transient user (users for whom only a cookie is stored in a database) and used this functionality to mimic the registration of a new user. Figure 10 shows the time taken to set up a new user for each framework compared to the corresponding standalone application. For this graph, we took the average response time for each endpoint and summed them to estimate the total duration of the registration phase. The results show that the registration phase of the standalone application is faster than the registration phase

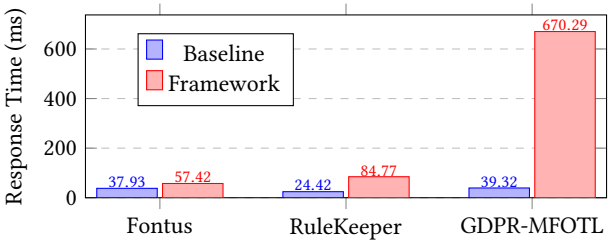


Figure 10: Response Time and Overhead of the Registration Phase for the Three Frameworks.

of the frameworks, with Fontus introducing the least amount of overhead and GDPR-MFOTL introducing the most.

Table 5 shows the response times for the registration phase for each framework and each endpoint that is called during the registration phase. Fontus is the only framework that calls a single endpoint to set up a new user, while RuleKeeper and GDPR-MFOTL need to call multiple endpoints. This is because we defined the */register* endpoint for Fontus as a source and sent the user's consent choice with a cookie. For both RuleKeeper and GDPR-MFOTL, we need to make additional calls to set the consent choice for the new user. Additionally, the registration phase for GDPR-MFOTL is the most time-consuming, with the */record* endpoint taking the most time to process. The issue with GDPR-MFOTL is that it was impossible to retrieve the taint value from the enforcement backend directly within the call to */register*. Consequently, we had to query the privacy dashboard to retrieve the taint-value for the user's personal data. GDPR-MFOTL also requires each user to have two accounts. We need to log in to (*/login*) GDPR-MFOTL before logging in to the application (*/1/login*). We did not find a way to insert a new user into the framework's database other than manually modifying the SQLite database. Therefore, we only measured the registration phase for GDPR-MFOTL starting from the moment the user was already registered with the framework. Overall, each of the frameworks adds a certain amount of overhead to the registration phase of BreezeBlogs: 51.39% for Fontus, 247.11% for RuleKeeper, and 1604.72% for GDPR-MFOTL.

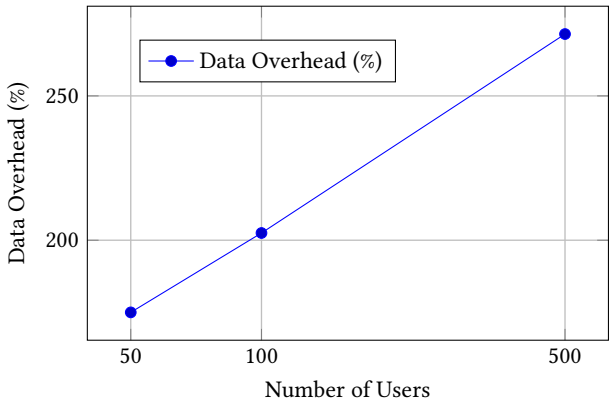


Figure 11: Data Overhead Analysis in Percentage for Different User Counts (GDPR-MFOTL without QuestDB).

Table 5: Endpoint Measurements for the Registration Phase.

Endpoint	Response Time (ms)
Fontus (baseline: 37.93 ms)	
/register	57.42
Total Response Time	57.42
RuleKeeper (baseline: 24.42 ms)	
/register	16.68
/consent/set-consent	68.09
Total Response Time	84.77
GDPR-MFOTL(baseline: 39.32)	
/1/login	32.31
/1/register	76.31
/login	14.57
/record	522.17
/record/add_consent	24.93
Total Response Time	670.2

Table 6: Storage Overhead Analysis for GDPR-MFOTL (without QuestDB).

#Users	W (kB)	W/o (kB)	O (kB)	O (%)
50	451	163	287	175.00
100	496	163	332	202.50
500	852	229	623	271.43

W: with framework, W/o: without framework, O: overhead

B.3 GDPR-MFOTL Overhead Without QuestDB

Table 6 and Figure 11 show the storage overhead without QuestDB, to get a better overview of the impact of different numbers of users. In this evaluation, we see a similar behaviour as for RuleKeeper, with the overhead accelerating more quickly for a larger number of users.