RuleKeeper: GDPR-Aware Personal Data Compliance for Web Frameworks

Mafalda Ferreira, Tiago Brito, José Fragoso Santos, Nuno Santos
INESC-ID / Instituto Superior Técnico, Universidade de Lisboa
{*mafalda.baptista, tiago.de.oliveira.brito, jose.fragoso, nuno.m.santos*}*@tecnico.ulisboa.pt*

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179395

***Abstract*—Pressured by existing regulations such as the EU GDPR, online services must advertise a personal data protection policy declaring the types and purposes of collected personal data, which must then be strictly enforced as per the consent decisions made by the users. However, due to the lack of system-level support, obtaining strong guarantees of policy enforcement is hard, leaving the door open for software bugs and vulnerabilities to cause GDPR-compliance violations.**

We present RuleKeeper, a GDPR-aware personal data policy compliance system for web development frameworks. Currently ported for the MERN framework, RuleKeeper allows web developers to specify a GDPR manifest from which the data protection policy of the web application is automatically generated and is transparently enforced through static code analysis and runtime access control mechanisms. GDPR compliance is checked in a cross-cutting manner requiring few changes to the application code. We used our prototype implementation to evaluate RuleKeeper with four real-world applications. Our system can model realistic GDPR data protection requirements, adds modest performance overheads to the web application, and can detect GDPR violation bugs.

1. Introduction

Protecting personal data has become a major concern for most online organizations. In particular, the EU General Data Protection Regulation (GDPR) [1] imposes strict access control requirements on data controllers when managing the personal data of their customers. To this end, data controllers must publish a human-readable policy declaring the personal data to be collected and the purposes for which it will be processed. Once consent is granted by data subjects, data controllers must comply with the agreed policy, otherwise, they may incur the payment of heavy fines [2–5].

However, it is not easy to enforce personal data protection policies in a full-blown web application. Developers often write their applications using full-stack development frameworks that are currently agnostic to the GDPR. For instance, with MERN [6], they write the application code in JavaScript using React.js [7] to implement the frontend, Express.js [8] and Node.js [9] the backend, and MongoDB [10] the database tier. Yet, MERN and other popular web frameworks provide no native support to help developers specify and enforce personal data protection policies, opening the door to GDPR compliance violations and privacy breaches.

To help developers fulfill the GDPR requirements for personal data protection, various system-level mechanisms have been proposed [11–19]. On the one hand, consent management platforms (CMP) [18, 20] focus on the application frontend, enabling cookie banners to be managed in a GDPR-compliant manner. Unfortunately, recent studies identified various legal violations in popular CMPs [18, 21–23]. CMPs are also intrinsically limited as they lack mechanisms to enforce data protection policies at the application backend. On the other hand, systems like Qapla [12] only act at the storage layer, rendering them unable to detect data access violations that depend on the application context, e.g., the notion of purpose or user authentication state. Riverbed [13] and PrivGuard [14] adopt a holistic approach, where they keep track of personal data ownership and user consent preferences throughout the entire data workflow. However, they focus on specialized usage scenarios, relying on mechanisms (e.g., trusted hardware) which are difficult or unsuitable to apply in typical 3-tier web applications built with full-stack frameworks, which are the focus of our work.

We present RuleKeeper, a GDPR-aware policy compliance system for web frameworks. Using MERN for demonstration, RuleKeeper incorporates cross-cutting GDPR compliance features whilst requiring minimal changes to the application code. With RuleKeeper, the data protection policy of the website is automatically generated from a machine-readable GDPR *manifest* written by the developer. This manifest specifies the personal data and purposes for which the web application can process the data. RuleKeeper will then enforce the consent decision of website users, preventing business-level operations from manipulating personal data for purposes that data subjects have not agreed to. Put simply, by approving the data policy of a website powered by RuleKeeper, "what you agree to is what you get".

To automatically detect and prevent GDPR compliance violations, RuleKeeper includes three main design novelties. First, to overcome the semantic gap between abstract GDPR concepts such as "personal data" or "purpose" and application's JavaScript code and MongoDB queries, RuleKeeper introduces a *domain-specific language* (DSL) for specifying the GDPR manifest. Our DSL features a small set of intuitive language constructs that allow developers to easily bridge this semantic gap without overwhelming them with unnecessary complexity or GDPR's legal terminology.

Second, to prevent GDPR violations, the JavaScript code implementing the business logic and the database queries must not be allowed to process personal data unless this

DOI 10.1109/SP46215.2023.00058

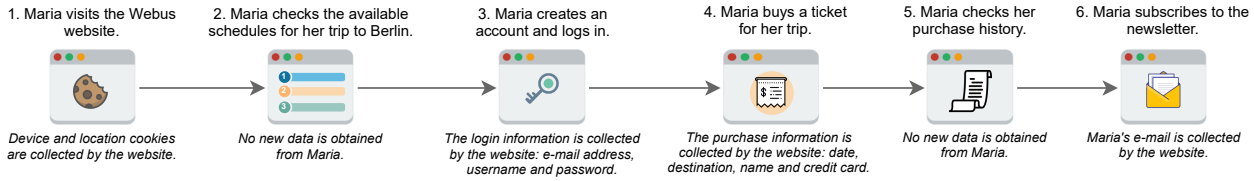

Figure 1: Example of user interaction with Webus: the caption below the screens tells the data collected at each step.

data is strictly used for the purposes indicated in the GDPR manifest. To perform these checks as the application codebase and data protection policy itself evolve, RuleKeeper includes a *static analysis tool* that automatically checks for the presence of GDPR compliance bugs. This tool generates a graph-based model of the JavaScript code and uses it along with the GDPR manifest to look for violations of GDPR's purpose limitation and data minimization guidelines.

Third, the system needs to efficiently keep track of the consent preferences for every user visiting the website and block any business-level operations that may attempt to access personal data against their will. Given that such access control decisions must be made at runtime, RuleKeeper includes a *dynamic policy enforcement middleware* that implements "sticky banners", i.e., an evolution of cookie banners where users' consent is recorded and their respective policy decisions are dynamically and transparently enforced at runtime. Our dynamic policy enforcement also adds another defensive barrier against external attempts to exfiltrate personal data by exploiting vulnerabilities in the application code. Moreover, it allows identifying any false negatives that may have potentially been missed by the static analysis, ensuring that our system's policy enforcement is sound, i.e., RuleKeeper can find all the compliance violations in scope.

We implemented an open-source prototype of RuleKeeper for MERN. To assess the expressiveness power of its policy language and its fitness for real-world scenarios, we conducted four case studies. One based on a real clinical analysis scenario, building the entire application from scratch, where we learned that RuleKeeper can be used to fully express data protection requirements in the healthcare domain. The others are based on large preexisting and popular applications, and allowed us to learn that RuleKeeper can detect compliance bugs in complex and evolving applications. We experimentally evaluated the performance of RuleKeeper, and found that the overheads incurred by the benchmark applications are acceptable given the added security and policy compliance benefits of our system.

In summary, this paper makes three main contributions:

1) The design of a GDPR-aware personal data compliance system for full-stack web frameworks, relying on a DSL for specifying GDPR requirements and a combination of static and dynamic analysis to enforce policy compliance.
2) A prototype implementation of the system for the MERN framework; we make our source code publicly available.
3) An evaluation of RuleKeeper using complex real-world applications demonstrating the expressiveness of our DSL, the effectiveness of our techniques at thwarting GDPR violations, and the performance of our system.

2. Motivation and Design Goals

2.1. Motivating Example

To motivate the need for our system, consider Webus, a hypothetical toy web application. It implements an online bus booking service that lets users browse bus schedules, buy tickets, see their purchase history, and subscribe to a newsletter. Figure 1 illustrates the steps performed by Maria as she interacts with Webus. To book a bus ticket to Berlin, as she lands on the website's front page (1), Maria must consent to the collection of cookies as per a data protection policy popping up in the screen. She then looks up for available trips (2), finds a suitable trip, and creates a user account by providing: email address, username, and password (3). After logging in, Maria buys her ticket, which demands her name and credit card information (4). To check if the purchase was successful, she visits her purchase history (5). Lastly, she subscribes to the Webus newsletter (6).

To abide by the GDPR, the Webus provider, acting as both *data controller* and *data processor*, must i) specify the purposes and means of personal data processing, and ii) implement technical measures to safeguard the protection of personal data (Articles 4, 24 and 28 of the GDPR [1]). To satisfy i), types and purposes of collected personal data must be explicitly declared in the policy. In this case, Webus collects multiple personal data items: e-mail address, username, name, credit card number, and ticket purchase history. These items are differently used for two main purposes: *ticket-managing purposes*, served by the operations "create an account", "buy a ticket", or "read the purchase history", and *marketing purposes*, which are put into action by operation "subscribe to newsletter". As for ii), enforcement measures must also respect the following GDPR guidelines:

- *Purpose Limitation*: Article 5.1 b) states that personal data must be collected for specified, explicit and legitimate purposes and must not be processed in a manner that is incompatible with those purposes. For instance, a credit card number collected for ticket managing purposes cannot be used for targeted marketing.
- *Data Minimization.* As per article 5.1 c), personal data should not be processed if not needed for the initially intended purpose. E.g., as the user's home address is not necessary for buying a ticket, it should not be collected.
- *Lawfulness of Processing.* Articles 5.1 a) and 6 declare that each purpose must have a valid lawful reason to process the personal data, such as the users' consent. Data subjects, such as Maria, are required to give their positive, specific, and unequivocal consent to each one

of the purposes for which they allow their data to be collected. For instance, if a data subject only gives their consent to ticket managing purposes, then Webus is not allowed to send him newsletters for marketing purposes.

System model: To build a web application like Webus while meeting the above GDPR requirements, we focus on classical 3-tier web architectures implemented with the help of popular full-stack frameworks such as MERN. In MERN, the presentation layer consists of web pages and JavaScript code running on the browser, and it is implemented with React.js. The logic layer consists of server-side code running on Node.js. With the assistance of Express.js, this code exposes an API for serving HTTP(S) requests to Webus operations by the presentation layer. The API is composed of a *route* per operation (i.e., a specific URL) and each operation is implemented by a piece of JavaScript code named *controller*. For instance, the "buy ticket" operation exposes route `/buy_ticket` and it is served by the controller listed in Figure 3. This layer also implements user authentication and access control functions, and processes the data from the data tier using object-relational mapping (ORM). Lastly, MERN's data layer uses MongoDB, a non-relational DBMS, for storing persistent data in databases. Each database is laid out via a *schema*. In Webus, the data is organized in four tables, containing the information suggested by their respective names: "Schedules", "Users", "Tickets", and "Newsletters". "Schedules" stores bus schedule information and is the only table that does not contain personal data.

2.2. Threats to GDPR Compliance

Using the Webus application, we identify several threats to GDPR compliance which we group into three categories: *compliance bugs* (example 1 and 2), *security vulnerabilities* (example 3) and *consent violations* (example 4).

Example 1: Data processed for incompatible purposes. Suppose Webus developers want to implement a new marketing feature to send promotional codes to frequent travelers ($\geqslant$ 10 trips in 2021) if they subscribe to the newsletter. So, besides the e-mail, the subscription logic will also need to process the ticket history as well. Figure 2 illustrates the new changes to the source code. Originally, since the "subscribe to newsletter" operation is bound to marketing purposes, it should only be allowed to process the e-mail of the subscriber. However, this new version will now access personal data that was not meant to be used for marketing purposes, i.e., the user's trip history. Given that there is no mechanism in place to validate GDPR compliance, this code upgrade will introduce a compliance bug leading to the violation of the purpose limitation principle.

Example 2: Reflective compliance bug. The Webus service had a problematic bus overbooking policy. So, the development team made a small code fix to revert it. Figure 3 shows the new code version: when a person buys a ticket for a trip, that information is added to the "Schedules" table; users can no longer buy tickets once the maximum capacity is reached. To avoid the mistake of example 1, developers verified that

```
function subscribe(req, res) {
 const { e_mail } = req.body;

 connection.query(`INSERT INTO newsletters (e_mail) VALUES
 ↪  ('${e_mail}')`, (err) => {

  /* New changes associated with the promo code */
  connection.query("SELECT * FROM tickets WHERE e_mail = '${e_mail}'
  ↪  AND year(date)>=2021", (err, tickets) => {
    if (tickets.length >= 10) sendPromoCode(e_mail);
    res.sendStatus(200); });
  });
}
```

Figure 2: Code snippet of examples 1 and 3. Compliance bug causing unlawful processing of personal data for marketing.

```
function seeSchedules(req, res) { /* Return all schedules from all
↪  time */
  connection.query("SELECT * FROM schedules", (err, schedules) => {
    res.status(200).json(schedules); });
}

function buyTicket(req, res) {
  const { name, credit_card, email, destination, date } = req.body;

 /* Add new ticket */
 connection.query(`INSERT INTO tickets (name, destination, date,
 ↪  creditcard, e_mail) VALUES ('${name}', '${destination}',
 ↪  '${date}', '${credit_card}', '${email}')`,(err) => {

    /* Add new traveler to trip - new changes */
    connection.query(`UPDATE schedules SET travelers =
    ↪  CONCAT(travelers, '${', ' + name}') `, (err) => {
        res.sendStatus(200); });
  });
}
```

Figure 3: Code snippet of example 2. This application bug leads to the unlawful processing of personal data.

the new "buy ticket" version follows the purpose limitation and data minimization guidelines. Unfortunately, they have overlooked an important fact. The "see schedules" operation was originally not consigned to a specific GDPR purpose as it does not process personal data. However, the same query that the `seeSchedules` function was previously invoking will now (unlawfully) return personal data about travelers' trips. Indirectly, a GDPR compliance bug was introduced that violates the two aforementioned principles.

Example 3: Purpose escalation attacks. A remote attacker may attempt to extract personal data from Webus by exploiting an SQL injection vulnerability inside vulnerable code that should not even have access to that specific kind of data. For instance, the "subscribe to newsletter" operation listed in Figure 2 is only expected to have access to the subscribers' emails. However, this code is vulnerable to an SQL injection. By typing in the e-mail field: `"" UNION ALL SELECT name, credit_card FROM tickets`, a remote attacker can leak the contents of columns "name" and "credit_card" from the "Tickets" table, which contain highly sensitive user identification and credit card information. We call this kind of attack a *purpose escalation attack*, where the attacker is able to access personal data inaccessible for the purpose(s) that the vulnerable function is associated with.

Example 4: Defective consent management. The Webus team used a third-party library to manage users' consent.

The library automatically pops up a cookie banner with a customized message indicating the requested personal data types and purposes (i.e., ticket booking and marketing). The library reads the user's response and saves a cookie on the browser's cache. Unfortunately, this type of libraries [24, 25] only relies on a cookie to remember the user's decision in future visits and does not enforce this decision on the server-side. So, even if a user consents to ticket booking purposes only, data can still be processed for marketing purposes.

2.3. Goals and Threat Model

Our goal is then to design a GDPR-aware policy enforcement system for web frameworks to help web developers thwart the aforementioned threats. We seek a solution that: i) allows specifying *expressive data protection policies*, thus serving the needs of various organizations and application scenarios, ii) is *transparent*, ensuring that policies are shown to the users in a clear, complete, and accurate way, reflecting how the application actually handles the data, and iii) is *easy to maintain*, detaching policy enforcement from application code. In this work, we target MERN, but we explain in Section 4 how our techniques can be applied to other web frameworks. Given the GDPR's extensiveness, we concentrate on ensuring compliance with the three GDPR guidelines indicated in Section 2.1. Section 8 further discusses how to extend our solution to cover other GDPR rules.

Threat model: We design our system as an aid for well-intentioned developers. It aims to help organizations and developers alike mitigate potential GDPR violations due to the accidental introduction of bugs in web applications (see Section 2.2). In this sense, RuleKeeper is not meant to defend against a malicious hosting organization or malicious developers willing to intentionally introduce GDPR compliance violations in the code. Developers will be using trusted APIs provided by the native web framework (MERN) and our system; we expect any potential bugs to arise exclusively from accidental programming errors. Additionally, we aim to protect against external adversaries that may attempt to exploit vulnerabilities in the application code causing data breaches through SQL injections, more specifically, via purpose escalation attacks as described in Section 2.2.

Assumptions: We assume that the developer(s) is(are) responsible for specifying a data protection policy for the target web application. This policy is expected to reflect the specific terms of the GDPR regulations that apply to the particular organization running the web application. To this end, the web developer(s) may require the assistance of the company's Data Protection Officer (DPO). As for the execution environment, our trusted computing base includes the browser runtime, the MERN software stack, and the components of our system. The distributed components communicate with each other using secure channels.

3. System Design

Figure 4 depicts a RuleKeeper deployment for a 3-tier web application. The yellow boxes represent the application-specific components that are present in a typical behavior of a 3-tier architecture system. The blue boxes represent RuleKeeper's specific software components. The system operates in two phases: *offline phase* and *runtime phase*.

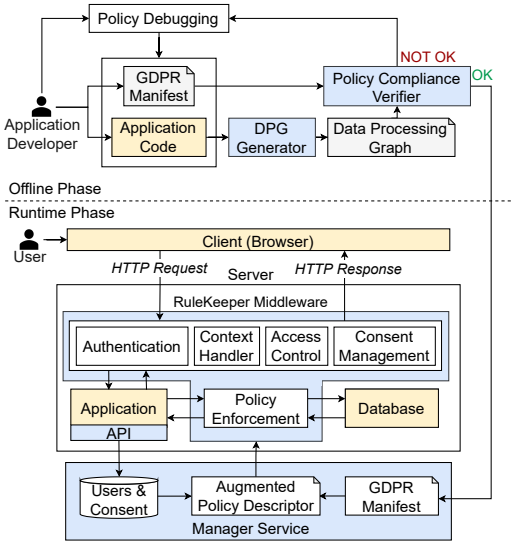

Figure 4: Architecture of a web application using RuleKeeper.

The offline phase takes place at development time before deploying the web application to production. In this phase, the web developer specifies a GDPR *manifest* from which the data protection policy will be generated. RuleKeeper implements a static code analysis pipeline that allows the web developer to verify if the policy reflects the way that the application behaves. This verification is performed in two steps. First, a code analysis tool generates a model of the application code which we designate Data Processing Graph (DPG). Then, a compliance verification tool will look for inconsistencies between the DPG and the GDPR manifest. If this is the case, the developer needs to debug either the application code or the GDPR manifest itself to resolve these inconsistencies. When the validation step passes, the GDPR manifest is loaded to RuleKeeper's runtime components.

In the runtime phase, RuleKeeper implements dynamic policy enforcement and consent management functions using two components: *middleware* and *manager service*. The middleware consists of application-linked libraries and plays two roles: i) keeps the manager server updated with user-related information, and ii) enforces dynamic access control and consent management validations. The manager service runs in a centralized server and coordinates the middleware. Importantly, it generates a data structure named Augmented Policy Descriptor (APD) which contains the information required by the middleware to dynamically enforce the policy, such as the GDPR manifest and user consent preferences. The middleware leverages this information in two main occasions. In one case, it intercepts the HTTP requests to verify the user's credentials and consent preferences, and enforce access control accordingly. The middleware also intercepts the database queries of the application to validate if they satisfy the GDPR policy and block them otherwise.

RuleKeeper supports dynamic policies in the sense that, whenever a meaningful contextual change occurs at runtime, e.g., a user changes their consent decision, the manager will accordingly update and propagate a new APD. Moreover, when a policy changes (e.g., due to a business-level decision), these changes can be reflected in a new manifest, checked through static analysis, and propagated by the manager via a new APD to the connected middleware instances. If the application is updated, e.g., with a feature that collects new personal data, the manifest needs to be updated, and the static analysis needs to be rerun to report mismatches.

3.1. Specifying Policies

To specify a GDPR manifest expressively and in a non-ambiguous way, a central challenge is to bridge the semantic gap between application and GDPR domains. Our approach is to create a simple DSL where the developer first specifies abstractions for two independent conceptual planes (*application* and *GDPR*) and then establishes *mappings* between planes. Next, we explain this idea using Figure 5 as a toy GDPR manifest that a developer would write for Webus.

Application plane: The developer starts by specifying several GDPR-agnostic attributes of the web application. The `DATA-ITEMS` field defines labels for all data types that can be collected and processed by the web application. In our example, Webus processes seven data types: tickets' buyer name and credit card, tickets' date and destination, schedules' date and destination, and newsletter e-mail. This data, whether personal or not, is processed by the application through specific operations that must be labeled and listed in the field `OPERATIONS`. Webus has four: "see schedules", "buy ticket", "see purchase history", and "subscribe to newsletter".

GDPR plane: The next step is to specify attributes related with the GDPR requirements in scope. Firstly, personal data must be tagged using the `PERSONAL-DATA` field. Of all the data types on Webus, only the schedule information is not considered personal. The purposes for which the data can be processed are defined in the `PURPOSES` field. Webus collects data for two distinct purposes: ticket management and marketing. The data that can be collected and further processed for each purpose is described using the `DATA-COLLECTION` field. Due to the data minimization principle, each purpose cannot process data that was not initially collected for it. Additionally, according to the lawfulness of processing principle, each purpose must be associated with a lawfulness base expressed by the `LAWFULNESS-BASE` field. In Webus, both lawfulness bases are the users' consent. Each operation is then associated with a purpose using the `EXECUTED-FOR` field. Webus "subscribe to newsletter" operation is executed for marketing purposes. So, for instance, if this operation tries to access the user's name, it will be marked as not compliant.

Mapping planes: To narrow the semantic gap between GDPR policies and application code, the `DATA-MAPPING` field maps the data items to the corresponding database schema, particularly the table and column where it is stored. Operations are mapped to the corresponding web endpoints exposed by the web application, using the `OPERATION-MAPPING`

```
# Application plane
DATA-ITEMS: ticket buyer name, ticket destination, ticket date,
↪  ticket buyer credit card, trip destination, trip date, email.

OPERATIONS: see schedules, buy ticket, see purchase history,
↪  subscribe to newsletter.

# GDPR plane
PERSONAL-DATA: ticket buyer name, ticket destination, ticket date,
↪  ticket buyer credit card, email.

PURPOSES: ticket management, marketing.

DATA-COLLECTION:
  ticket buyer name, ticket destination, ticket date, ticket buyer
  ↪  credit card ARE COLLECTED FOR ticket management purposes.
  email IS COLLECTED FOR marketing purposes.

LAWFULNESS-BASE:
PURPOSE ticket management HAS LAWFULNESS BASE consent.
PURPOSE marketing HAS LAWFULNESS BASE consent.

EXECUTED-FOR:
  buy ticket, see purchase history ARE EXECUTED FOR ticket
  ↪  management purposes.
  subscribe to newsletter IS EXECUTED FOR marketing.

# Mapping planes
DATA-MAPPING:
ticket buyer name IS IN COLUMN name OF TABLE tickets.
ticket destination IS IN COLUMN destination OF TABLE tickets.
ticket date IS IN COLUMN date OF TABLE tickets.
ticket buyer credit card IS IN COLUMN credit_card OF TABLE tickets.
trip destination IS IN COLUMN destination OF TABLE schedules.
trip date IS IN COLUMN date OF TABLE schedules.
email IS IN COLUMN e_mail OF TABLE newsletter.

OPERATION-MAPPING:
see schedules IS MAPPED TO ENDPOINT GET /schedules.
buy ticket IS MAPPED TO ENDPOINT POST /buy_ticket.
see purchase history IS MAPPED TO ENDPOINT POST /purchase_history.
subscribe to newsletter IS MAPPED TO ENDPOINT POST /subscribe.

DATA-OWNERSHIP:
OWNER IN TABLE tickets IS IN COLUMN name.
OWNER IN TABLE newsletters IS IN COLUMN e_mail.
```

Figure 5: An example of a RuleKeeper policy for Webus.

field. To process personal data with regard of whom it belongs to, e.g., to verify consent preferences, the `DATA-OWNERSHIP` field associates each database table with the embedded column that represents the owner of the data (i.e., data items present in a given row belong to the data subject identified by the ownership column). In Webus, column "name" identifies the owner of the data in each row of the "Tickets" table and the "e_mail" column tells the owner of the data of the "Newsletters" table. At runtime, RuleKeeper will also need to rely on information provided by the application to make complete access control decisions, e.g., the identity of a currently authenticated user.

3.2. Validating Policies with Static Analysis

Web developers use RuleKeeper in the offline phase to check if the application code satisfies the restrictions specified in the GDPR manifest (e.g., check for compliance bugs in the implementation of "buy ticket"). RuleKeeper verifies this using a static code analysis pipeline. After giving an intuition of our approach, we describe each pipeline stage.

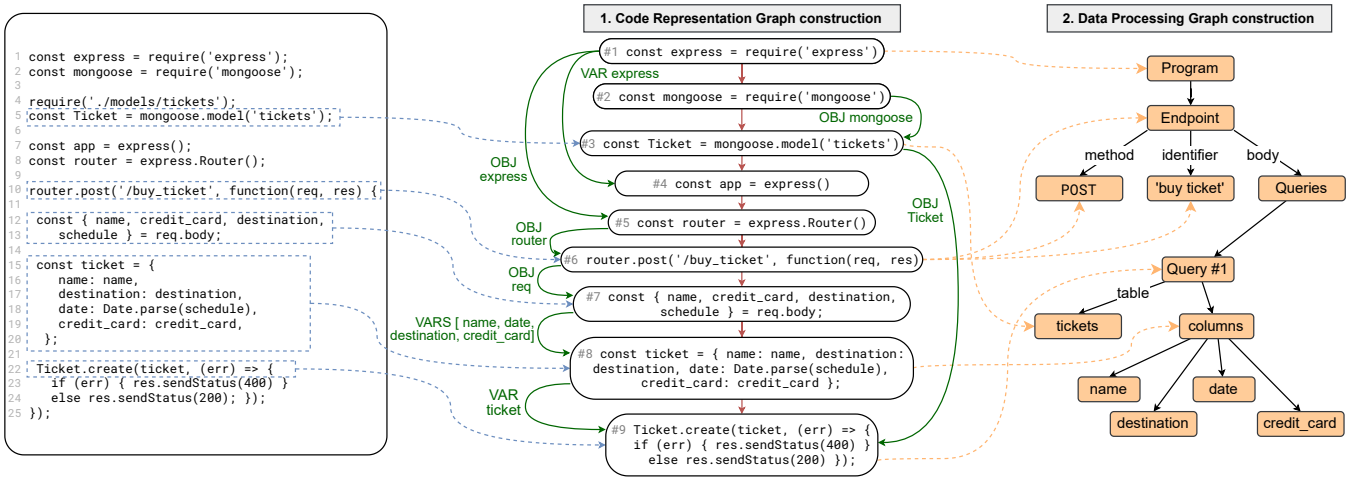

Figure 6: Static analysis stages for Webus sample code: the program is modeled into a CRG, which is then processed into a DPG.

Approach overview: Our approach is divided in two steps: first, we generate a model of the web application code which we call Data Processing Graph (DPG), and then we use the DPG to search for inconsistencies in the manifest. The DPG aims to automatically create a picture of the web application that can tell us: i) all the operations that exist in the code, and ii) all the data types accessible to each operation. Based on this information, we can then check if operations' data accesses are legal by comparing them against the respective specification in the GDPR manifest. Figure 6 showcases a DPG sketch that reflects a toy implementation of the "buy ticket" operation pertaining to Webus. In a nutshell, the DPG: i) identifies each operation with an API endpoint and the code associated with it (e.g., `/buy_ticket`), and ii) identifies the accessed data types based on the database queries performed by the code at the endpoint (e.g., through `Ticket.create`). The static analysis automatically extracts the endpoints and data queries executed at those endpoints based on an intermediate representation that we call Code Representation Graph (CRG). Next, we explain how these data structures are generated during code analysis, and then describe the algorithm for compliance checking.

1. Generating the Code Representation Graph: To build the CRG, we adapted the idea of code property graphs [26–28] to server-side JavaScript, with support for object dependencies. Thus, the CRG combines the abstract syntax tree (AST), the control flow graph (CFG), and the program dependency graph (PDG) in a single graph structure. In the CRG of Figure 6, the AST nodes are represented with numbered rounded boxes. For simplicity, we omit the AST edges and summarize the information in *Statement* nodes – see [29] for the complete CRG. The CFG and PDG edges between AST nodes are represented in red and green, respectively. First, RuleKeeper parses the JavaScript code and builds the corresponding AST. The AST is then traversed to produce the CFG, which creates edges between consecutive code statements and to possible branching operations – `if`, `while`, etc. Finally, the CFG is traversed to determine data dependencies between code statements, producing a PDG. A data dependency can be classified as variable dependency (`VAR`) or object dependency (`OBJ`). To clarify the differences, consider the example in Figure 6. The program declares the variable `express` (line 1) and variable `app` (line 7), which depends on the former. Thus, our analysis creates a data-dependency edge between these statements with the label `VAR express`. On the other hand, the statement at line 8 declares a new variable `router` that also depends on `express`, but this time the content of the `router` variable does not directly depends on the value of the variable `express`, but instead on the value of a property of the `express` object. Consequently, a data-dependency edge between these statements is created with the label `OBJ express`. This analysis is intra-procedural. The output of this step is the CRG, which can then be queried using a graph database querying language to extract information for building the DPG.

2. Generating the Data Processing Graph: To build the DPG, we import the CRG into a graph database to allow for graph traversals, i.e., queries to check how data is being processed within the application. Next, we give an overview on the algorithm used to extract relevant information from the CRG using Figure 6 as an example. Our queries expect that the application implements endpoints and database queries using Express.js and Mongoose API calls.

1) First, we query the CRG for the registered endpoints, by searching code statements that register routes through function calls to Express.js's `Router` object. In the example of Figure 6, the query finds the statement that registers the `POST` endpoint (node #6) and checks if it depends on the `Router` object (node #5). Then, we inspect the AST to extract both the path for the endpoint `/buy_ticket` – first argument of the call in node #6 – and the callback function – second argument. This callback is executed when the `/buy_ticket` endpoint is accessed and contains statements that handle user data.
2) Next, we query the CRG for the database queries executed in the context of each endpoint. First, we search for patterns that correspond to `model` calls (node #3), which depend on the Mongoose module (node #2),

and extract the model name (`tickets`) by inspecting the AST. We then search for calls using the Mongoose Query API, such as `create` (node #9), that: i) depend on the model call identified before (node #3), and ii) are made inside the scope of the callback registered to the endpoints, such as `/buy_ticket` (node #6).

3) Finally, we search the AST of the Mongoose query calls for the data accessible (read or written) by the query. In this example, the inserted data corresponds to the first argument of the create call (variable `ticket`). For other Mongoose queries, e.g., `findOneAndUpdate`, the inserted data is in the second argument position. Since `ticket` is an object, we know that Mongoose will insert its properties into the database. With this information, we can generate a DPG for the policy compliance verifier. The DPG clearly tells that columns `name`, `credit_card`, `destination`, and `schedule` of table `tickets` are handled whenever the `POST` endpoint with path `/buy_ticket` is accessed, as shown in Figure 6.

3. Checking the DPG for policy compliance: From the resulting DPG, our analysis checks if the GDPR manifest reflects how the application processes personal data. The policy compliance verifier parses the manifest into a DPG-like data structure that associates endpoints with the personal data that the endpoints are allowed to process (e.g., endpoint `POST /subscribe` is associated with column "e_mail" of table "newsletters"). This way, one can directly match the information in the manifest with the DPG. At this point, the policy compliance analysis is divided in three validations:

1) *Personal data processing:* Looks for operations that process personal data but have not been declared in the manifest. To this end, RuleKeeper uses from DPG a filtered list of endpoints that process personal data.
2) *Purpose limitation:* Screens purpose limitation violations. RuleKeeper uses DPG, filtered with queries that process personal data, and checks if any personal data item is processed by an operation for a purpose that is not represented in the DPG-like manifest data structure.
3) *Data minimization:* Similar to 2), but checks if any operation for a given purpose processes more data than the represented in the DPG-like manifest data structure.

4. Debugging inconsistencies: If RuleKeeper's static analysis endorses the GDPR manifest, showing that it reflects the way that the application is processing personal data, then the application is ready for deployment. Otherwise, the developer must fix the detected inconsistencies. These bugs can either stem from an over-permissive application code or from an inaccurate GDPR manifest. Our tool reports them by indicating, for each endpoint: which data is expected to be processed (i.e., declared by the manifest), which data is actually being processed by the application, and which of the previous three validations is failing. The developer can either opt to i) update the application code, if she considers the code is over-permissive (i.e., processing more data than acceptable), or ii) update the policy to match the application code. If i), she simply needs to remove the unnecessary personal data from the query. If ii), she can: (1) update the `DATA-COLLECTION` rule in the manifest to include the personal data, if she considers the purpose of the operation, defined in the `EXECUTED-FOR` rule to be correct, or (2) update the `EXECUTED-FOR` rule, to amend the purpose of the operation. Other bugs may arise from inaccurate mappings, which the developer must also examine and fix if necessary.

`new_user(string user_id, string pass_hash, string role, string owner_id, string gdpr_role) ⇒ ()`
Creates a new user with a password, roles and the entity it represents.
`authenticate(string user_id, string pass_hash) ⇒ (string token)`
Authenticates a user and returns an authentication token.
`remove_user(string user_id) ⇒ ()`
Removes user.

Table 1: RuleKeeper's user-related data management API.

3.3. Enforcing Policies at Runtime

RuleKeeper relies on a complementary policy enforcement mechanism at runtime for two reasons. First, access control may change depending on individual users' consent to the data protection policy. Users may even selectively give their consent per purpose, which means that policies end up being dynamic. Second, due to limitations of static analysis in tracking program execution state, validating user requests at runtime constitutes the second and definitive line of defense for preventing GDPR violations. Next, we describe RuleKeeper's dynamic policy enforcement, starting by explaining how users give consent to the privacy policy.

Sticky banners: RuleKeeper displays a popup message to website visitors telling the purpose of the operation(s) and the data that will be collected. This popup – named *sticky banner* – is automatically generated by RuleKeeper off of the website's GDPR manifest. RuleKeeper will then record the user's preference (i.e., give/deny consent) and block all operations for which consent has been declined. Usability wise, users may feel annoyed if prompted too frequently for every operation. In contrast, showing a banner a single time asking for permission to collect all personal data types processed by the website may be overly permissive. To cope with this well-known security-usability trade-off, RuleKeeper lets web developers configure sticky banners with three display options: i) *per-site*: shows the sticky banner to the user only once presenting the website's complete policy, ii) *per-operation*: shows the banner for every new operation that processes personal data, and iii) *per-purpose*: shows the banner once per purpose, as the website requests access to personal data for a specific purpose for the first time. Sticky banners can be merged with cookie banners in the *per-site* scenario, but cannot for other display options, as sticky banners may need to be shown as the user navigates deeper into the website.

Managing users: Besides specifying the sticky banner option, developers must help RuleKeeper associate consent preferences to users. Remembering the preferences of each user is essential by the time the middleware needs to make access control decisions. To this end, RuleKeeper's middleware contains a built-in authentication mechanism that exposes a user management API (see Table 1) which

developers should use to create, authenticate, or remove users. This API provides an interface to the manager service, which stores user-related information on a local database, including user consent preferences as described next.

Tracking users' consent preferences: When a user wants to perform an operation, RuleKeeper's middleware intercepts the HTTP(S) request and checks if the user has already given their consent for the operation's purpose. If so, the operation proceeds; otherwise, the corresponding sticky banner pops up, only letting the operation continue if a positive consent is given. To keep track of users' past preferences, RuleKeeper stores positive consent records in the "Users & Consent" table and sends a *consent cookie* to a user's browser by the first time the user submits their consent preferences. The way this cookie is linked with a user's identity and consent preferences depends on two cases:

- *Transient users:* Users may need to share their personal data only within the context of a single web session. E.g., Webus allows users to buy tickets without creating an account. As no account was created using the `new_user()` API call (see Table 1), RuleKeeper uses the consent cookie as user identifier for as long the session lasts.
- *Authenticated users:* When users create an account and log into the website, RuleKeeper maps the consent cookie to the current user identity by invoking an API call: `authenticate()`. It works independently of the authentication mechanism itself, as it only needs to be invoked after a successful user authentication. An authentication token enclosed in the consent cookie serves as user identifier.

To avoid frequent queries to the "Users & Consent" table (see Figure 4), we enclose the list of purposes approved by the user in the consent cookie. Therefore, every time the user interacts with the website, RuleKeeper is able to extract the user's consent preferences from the cookie itself.

Runtime policy enforcement mechanisms: To make access control decisions at runtime, RuleKeeper relies on two separate hooks: *HTTP hook* and *DB hook*, both application-independent. The HTTP hook is located at the application entry point: it intercepts HTTP requests from the user's browser and validates the request's context. The DB hook is located at the application/database boundary, intercepting and validating the database queries. Next, we explain how these hooks work as they intercept and process user requests:

1. Intercepting an incoming HTTP request: RuleKeeper first needs to maintain some context ready before the policy can be evaluated. To this end, the HTTP hook first identifies the user, whether through authentication or a session identifier. Then, it associates his consent preferences. RuleKeeper automatically detects the operation, by combining the URL and method of the HTTP request. When both the user and operation are identified, it performs access control validations based on information provided by the application.

2. Intercepting a database query request: This hook intercepts database `write` query requests and performs two steps. First, it validates the query based on the owner of the data being queried. To find the data owners, RuleKeeper uses the `DATA-OWNERSHIP` rule and performs an extra query

1. for all $d \in pdata$ exists $p_d \in$ `DATA-COLLECTION`(d) and exists $p_o \in purposes$, such that $p_o \in p_d$
2. for all $d \in pdata$ and $p_o \in purposes$, $d \in$ `DATA-COLLECTION`(p_o)
3. for all $d \in pdata$ and $p_o \in purposes$, if `LAWFULNESS-BASE`(p_o) is $consent$, then for all o in `owners`(t), $p \in$ `consent`(o)
where $pdata \leftarrow$ `DATA-MAPPING`$(d) \in$ `PERSONAL-DATA` $\wedge$ $purposes \leftarrow$ `EXECUTED-FOR(OPERATION-MAPPING`(e)`)`

Figure 7: Policy evaluation conditions.

applying the same conditions as the original, which returns the value stored in the ownership column. To reduce overhead, RuleKeeper first checks if one of the columns of the query is the ownership column; if it is, it promptly returns that value. Secondly, RuleKeeper makes an access control decision based on the algorithm sketched in Figure 7. Given an endpoint e that processes data d, e will only be allowed to be executed by the user u if three conditions are satisfied: 1 and 2 validate purpose limitation and data minimization; 3 validates the lawfulness of processing (i.e., consent validation). This nomenclature reflects the DSL primitives specified in Section 3.1, with the exception of *owners*(d), which represent data subjects owning queried data d, and *consent*(o), which denotes the consent of each data owner. In case of a positive authorization decision, the request continues to the application, otherwise it is denied.

3. Intercepting a database query response: This hook intercepts database `read` query responses and applies the same logic as above. `read` queries must be intercepted after the query is executed to obtain with precision the data that is trying to be accessed. By contrast, `write` must be intercepted before the query is executed to prevent it from performing modifications of personal data without permission.

3.4. Policy Enforcement Properties and Limitations

Policy enforcement relies on a combination of static and dynamic analyses employed at development time and at runtime, respectively. Together, they aim to prevent GDPR compliance violations in a web application based on the policy specification given in the GDPR manifest. Next, we discuss the main properties of our policy enforcement techniques, focusing separately on precision and soundness.

Precision: Precision weighs the number of false positives in reporting inconsistencies between the web application and the GDPR manifest. As discussed in prior art [27, 30, 31], due to the dynamic nature of JavaScript, performing a precise and accurate static analysis of JavaScript-based web applications is difficult and may lead to misclassifications. Likewise, RuleKeeper's static analysis may report false inconsistencies between application code and GDPR manifest. For instance, CFG and DPG may contain edges that reflect implicit hidden flows; hence, the resulting CRG may contain seemingly violating edges that may never get triggered in practice. (In Section 8, we examine the obtained false positives while testing RuleKeeper with real-world applications.) Pruning out false positives requires manually analyzing the application to check if its code is fully compliant with the

manifest. Nevertheless, despite the added effort to the developers, RuleKeeper's static analysis brings two key benefits. First, it acts as a pre-filter [30] to preemptively identify true violations that would otherwise be detected at runtime only. Such a deferred detection (and consequent blocking) could impair service availability and increase maintenance costs. Second, RuleKeeper's static analysis can be used as a debugging tool to help developers reason about the privacy implications of their code and early-detect compliance bugs, as we demonstrate in the user study presented in Section 7.

Soundness: A sound analysis eliminates false negatives, i.e., RuleKeeper will not fail to report an existing inconsistency between application and GDPR manifest. This is the most critically-desired property for RuleKeeper, as it must not tolerate GDPR compliance violations to go past undetected. Unfortunately, RuleKeeper's static analysis is not sound, and thus can have false negatives. For instance, it relies on the AST to learn the path and the callback function of each endpoint of the target application. However, this information may not be statically accessible in the AST; for example, in one use case application studied in Section 6, the path is the return value of a function invocation and not a static string, rendering it impossible to annotate the DPG with endpoint information for this application. JavaScript is also known for its dynamic code generation capabilities, such as the `eval` function, which can result in missing function calls and lead to an incomplete CFG. In such cases, the resulting CRG may lack some edges that can result in violations of the GDPR manifest. To prevent false negatives, we leverage RuleKeeper's existing runtime infrastructure with a dynamic policy enforcement mechanism (see Section 3.3). At runtime, RuleKeeper can intercept all the concrete execution flows and monitor the observable violating flows skipped by the static analysis. Therefore, complementing static with dynamic analysis guarantees soundness, ensuring that all non-compliant data processing flows are detected as long as the manifest is complete. If the manifest is incomplete, e.g., missing the definition of a personal data type, the verified properties may not be the ones intended by the developer. To avoid this problem, manifests can be specified with the aid of automatic personal data classification tools [32], cross-validation by several developers, and cooperation with DPO.

4. Implementation

We implemented a full prototype of RuleKeeper. Our code is available open source [29]. The middleware was written in JavaScript for Node.js v14.16.1 + Express v4.16.1 and MongoDB Community Server v5.0.2, supported by Mongoose v5.13.9 ORM [33], with 1095 lines of code (LoC). To intercept HTTP requests, we use Express built-in application-level middleware. To intercept MongoDB queries, we use Mongoose global plugins. As for developers' coding style, we only expect a proper use of Mongoose and Express API calls as per the prescribed documentation.

We deployed the policy engine inside the middleware using Open Policy Agent v0.36.0 (OPA) [34], an open source policy engine that enforces Rego [35] policies. Rego extends Datalog and allows for assertions on structured data stored in JSON documents. OPA makes context-aware policy decisions in the middleware by querying a Rego-compiled policy that encodes our GDPR compliance conditions along with JSON documents that contain both the dynamically fetched data from the manager and the manifest. We integrate OPA with RuleKeeper using the OPA WebAssembly [36] module.

The manager was written as a Node.js v14.16.1 + Express v4.16.1 and MongoDB Community Server v5.0.2 web application, with 1461 lines of JavaScript and TypeScript code. Communication uses Socket.IO v4.3.1 [37] events.

We implemented RuleKeeper's static analysis tool using the Esprima v4.0.1 [38] parser with 2111 lines of JavaScript code. This analysis outputs the graph's nodes and edges that are later imported to a graph database. We used Neo4j v4.2.1 [39] as the graph database engine and a custom Python script, with 390 lines of code, to execute 11 custom queries that extract relevant information from the graph.

Portability: Although we target MERN for its popularity [40, 41], RuleKeeper relies on two general techniques that can be adapted to web stacks exposing similar programming abstractions, i.e., model-view-controller and REST APIs: (i) DPG creation in the static analysis, and (ii) middleware hooks in the runtime analysis. For JavaScript-based web frameworks, porting RuleKeeper would require the adaptation of these techniques to new database and hook interfaces. For other programming languages, e.g., PHP, static analysis could leverage already existing tools [42, 43].

5. Case Studies

To assess the expressiveness of RuleKeeper's policy language and its fitness for real-world scenarios, we conducted four case studies: one where we built the web application from scratch, and the others based on legacy applications.

5.1. GDPR Compliance in LEB

We collaborated with the clinical laboratory *LEB - Laboratórios Elisabete Barreto* [44] to develop a prototype intranet service for supporting its internal business processes. Our main goal was to study how well RuleKeeper is able to accommodate GDPR data protection requirements of such a privacy-sensitive application space as the healthcare domain.

GDPR requirements for clinical laboratories. APAC [45], a Portuguese association for clinical analysts, developed a turnkey GDPR framework that establishes guidelines for achieving GDPR-compliance in clinical laboratories such as LEB. To ensure that LEB's intranet service follows these guidelines, we extensively analyzed APAC's impact assessment reports from which we drew essential information for the specification of LEB's manifest, such as the data types that constitute personal data and the purposes of usage.

Prototype implementation. We analyzed LEB's internal processes that manage personal data. Then, for simplicity, we implemented a MERN-based intranet prototype focusing on the *pre-analytic process*, which is responsible for patient

registration and specimen processing. It supports three types of users: patients, receptionists, and system administrators. We implemented seven controllers (see Appendix A) for supporting the operations: (i) patient registration by receptionists, (ii) patient data handling by both receptionists and patients and (iii) user management by system administrators.

Code integration. Integrating RuleKeeper with LEB's prototype was relatively simple. First, we imported RuleKeeper's middleware with 3 lines of code (LoC) and integrated RuleKeeper's user management API calls, adding a total of 5 calls. Then, we deployed RuleKeeper manager and specified LEB's GDPR manifest with our DSL.

GDPR manifest. LEB had in place a human-readable privacy policy and a specific document to inform their patients on how LEB a) processes personal data and b) applies data protection principles, complying with Articles 12, 13, and 14 of the GDPR [1]. To write LEB's GDPR manifest, we combined information from these documents with APAC's impact assessment reports and LEB's internal processes. In total, we specified 66 DSL statements, including 11 personal data types, 2 purposes, and 10 operations. From this exercise, we verified that RuleKeeper's DSL is expressive, allowing us to fully specify a non-trivial, real-world policy covering the GDPR guidelines in scope (see Section 2.1).

5.2. Compliance in Legacy Applications

To study the challenges of retrofitting complex applications to work with RuleKeeper, we browsed popular open-source web applications in GitHub. From the 37K Express.js projects found, we filtered the most popular (>1K stars), resulting in 49 candidates, and further selected those processing personal data, obtaining eight projects. Lastly, we picked the three most elaborate applications regarding their data model/operations. Habitica [46] is a task manager with over 1M downloads in Google Play [47] and 9k GitHub stars. It is our most complex use case, comprising 28K lines of server-side code maintained for +10 years by 715 contributors. Amazona [48] is an Amazon-style, e-commerce application where users can order and review products (1.3k GitHub stars), and Blog [49] (3.3k GitHub stars) is a blog application for posting and commenting on articles. Both are simpler than Habitica but manipulate other personal data types that are likely to also be used in other applications. Next, we report our experience on integrating RuleKeeper with these applications, underlying the most complex case.

Habitica. This application is a habit building program where users complete tasks that represent real-life goals. We studied its data model and source code based on our knowledge of its workflow. Habitica's database contains 15 collections, including "Users" and "Tasks", which we considered the most relevant. The "Users" collection contains information about Habitica's users, such as authentication, purchases, and preferences, comprising over 300 fields of data. Habitica's API includes over 200 routes. We survey some of the operations that we considered most relevant in Appendix A.

Code integration. Table 2 lists the added lines of code to integrate RuleKeeper's middleware in each application. The modified application versions can be downloaded from [29]. Habitica required more effort than others (10 LoC added), since its code does not follow the typical structure of Node.js + Express.js applications. Blog did not require authentication calls since RuleKeeper natively supports its authentication mechanism (passport with local strategy [50]). Overall, we consider this effort low when compared with the current alternative, i.e., manually adapting the application code to give similar guarantees as RuleKeeper's, without bugs.

Application	DSL Statements				Lines of Code
	Personal Data	Purposes	Operations	Total	
LEB	11	2	10	66	8
Habitica	14	3	21	168	10
Amazona	13	3	15	161	7
Blog	11	2	18	127	6

Table 2: GDPR manifest and implementation effort.

GDPR manifest. To write a GDPR manifest for each legacy application, we studied their existing privacy policies and combined this information with our obtained knowledge of the source code. Table 2 shows that the number of DSL statements needed to write applications' respective manifests is quite manageable. In Habitica, the most complex case, the manifest comprises 168 statements in total, covering 14 relevant personal data types and 21 operations. Here, we considered as personal data the data directly associated with users in the collection "Users", and defined the purposes as member-, group-, and task management. In all applications, similar to the LEB case study, RuleKeeper's DSL can fully express the data protection requirements in our scope.

6. Experimental Evaluation

Our evaluation aims to answer the following questions:

1) What is the cost of RuleKeeper's runtime processing? We find that the overhead added by runtime hooks is 1.21× on average, dominated by the policy enforcement logic.
2) What is the client-perceived latency introduced by RuleKeeper? We measured the latency from the perspective of Web clients and observed a 13% increase in average latency when using RuleKeeper on legacy applications.
3) What is RuleKeeper's impact on the number of requests per second that a web application can sustain? We observe Habitica's behaviour when saturated and find it sustains less 11.9% of the requests in the worst case.
4) What is the impact of RuleKeeper in resource usage? CPU and memory usage increase, on average, in 4.84% and 3.87% respectively, in a saturated system.

Experimental setup. Our testbed consisted of four 64-bit Ubuntu 18.04.5 virtual machines (VMs) with 32GB of RAM and an 8-core Intel Xeon E5506 2.13GHz CPU deployed on different physical machines interconnected by a 1Gpbs LAN. VM1 is configured with our use-case web applications (LEB, Habitica v4.189.0, Amazona, and Blog) running as a Node.js + Express server. VM2 runs a MongoDB Community Server v5.0.2 database storing application data. VM3 hosts the RuleKeeper Manager, and VM4 runs the

Tid	Task	Query	Entity
L1	As a patient, access own patient data.	Read	Data Subject
L2	As a patient, update own patient data.	Update	Data Subject
L3	As a recepcionist, access a patient's data.	Read	Controller
L4	As a recepcionist, update a patient's data.	Update	Controller
L5	As a recepcionist, access several patient data.	Read	Controller

Table 3: User tasks used for benchmarking LEB.

App	Server Latency			Client Latency			Throughput			Efficiency	
	5th	avg.	95th	5th	avg.	95th	min.	avg.	max.	cpu	mem.
L	1.08×	1.26×	1.34×	1.06×	1.20×	1.25×	0.66×	0.74×	0.94×	7.40%	4.23%
H	1.14×	1.16×	1.19×	1.09×	1.10×	1.12×	0.88×	0.91×	0.94×	3.67%	7.15%
A	1.07×	1.14×	1.21×	1.04×	1.14×	1.26×	0.70×	0.85×	0.96×	2.30%	1.32%
B	1.25×	1.26×	1.27×	1.07×	1.15×	1.22×	0.70×	0.74×	0.77×	6.00%	4.58%

Table 4: Summary of RuleKeeper's experimental evaluation.

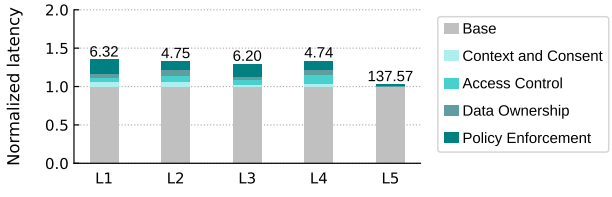

Figure 8: User task latency overhead in LEB, normalized to Base. Labels show RuleKeeper absolute latency in ms.

client-side experiments with up to 64 client threads running simultaneously. For LEB, we measured different user tasks whose parameters may influence RuleKeeper's performance (e.g., whether a task involves a *read* or an *update* query). Table 3 lists these tasks (*Lx*). For the other applications, we picked three tasks representative of frequent user activities: for Habitica, we considered *H1:* get user profile, *H2:* create a new task and *H3:* score an existing task; for Amazona, *A1:* check available products, *A2:* order a product and *A3:* check my orders; and for Blog, *B1:* get open articles, *B2:* get my blog profile and *B3:* comment on an article. We used *wrk* [51] to simulate traffic by generating HTTP requests for each one of the user tasks for 10 minutes each.

Server-side latency. Table 4 presents a summary of our results. To gauge the server-side overheads of RuleKeeper, we measured the total execution time of the selected user tasks for each application without RuleKeeper (*Base*) and with RuleKeeper, on the server side, reporting the arithmetic mean and 5th/95th percentiles. As shown in the first column of Table 4, the relative overheads introduced by RuleKeeper's runtime hooks in legacy applications range from 1.07 to 1.27×. For the legacy applications, Blog presents the highest overhead (1.27×) because tasks *B2-B3* perform several short queries that increase the relative contribution of RuleKeeper's runtime hooks to the total execution time of each task. This effect is also visible in LEB, where each task is very simple and only contains basic database queries, leading to a similar increase in the relative overhead. Overall, RuleKeeper only introduces an average absolute latency of 1.6ms per query. Given that in LEB each task performs a single query (see Table 3), we can use it to better understand the source of RuleKeeper's overheads. Figure 8 showcases

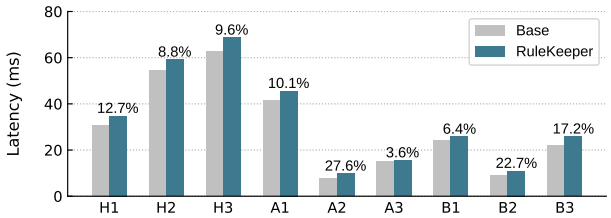

Figure 9: Average client-perceived latency in legacy application tasks. Labels show the relative overhead, in percentage.

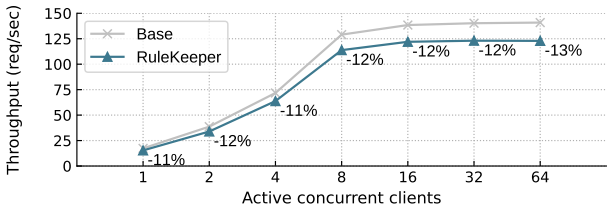

Figure 10: Habitica task scoring throughput.

LEB's overhead per task normalized to base, broken down into four logical components performed by RuleKeeper's runtime hooks. In general, these overheads are dominated by the execution time of the OPA WebAssembly module responsible for policy enforcement, including access control decisions. In *L5*, we observe a lower relative overhead because this task is more expensive as it involves the data of several patients, making the overhead less noticeable.

Client-side latency. We measured the response latency from the perspective of a web client (i.e., requests' RTT), excluding web page loading and rendering. Figure 9 presents the average latency of the legacy applications with RuleKeeper (blue bars) and without RuleKeeper (grey bars). On average, the client-perceived latency increase for legacy applications is 13.2%. In absolute terms, these are small client-perceived overheads and likely unnoticeable to Internet users.

Throughput. We also study how RuleKeeper affects the throughput. For Habitica, since most user actions are based on completing tasks, we used *wrk* to measure the amount of scored tasks (*H3*) that it can withstand per second, varying the number of concurrent clients (generating requests at a constant rate) until the server is saturated. Figure 10 reports the average *H3* requests per second without RuleKeeper (grey series) and with RuleKeeper (blue series) as the number of concurrent clients increase from 1 to 64; a rate of 140 requests per second sufficed to saturate the CPU. We can see that, with RuleKeeper, Habitica responds to 11.9% fewer requests when saturated. Measuring the throughput also for *H1-H2*, and computing the average across of *H1-H3* gives approximately a 9.3% throughput reduction for Habitica, as reported in Table 4. This table also shows the throughput reduction for the other applications, computed similarly to Habitica. This reduction reflects the extra processing cost added by RuleKeeper, which is the price to pay for better security and policy compliance guarantees. Blog and LEB present the highest throughput reduction for the reasons mentioned in the "server-side latency" paragraph.

Resource utilization. We measured the CPU and memory usage in saturation. We stressed RuleKeeper for saturation settings of 215, 96 and 138 requests per second (reqs/sec) for Habitica tasks (respectively), 24, 190 and 108 reqs/sec for Amazona's, 95, 265 and 83 reqs/sec for Blog's and 429, 431, 419, 429 and 7 reqs/sec for LEB's. Table 4 presents the mean of CPU usage and memory usage, using *dstat* [52]. On average, RuleKeeper modestly increases CPU usage in 4.84% and memory usage in 3.87%. The RuleKeeper manager is lightweight as it consumed on average 0.05% of CPU and 41MB of memory on standby, and 0.1% of CPU and 45MB of memory when receiving 1000 requests/hour.

7. Usability Study

To assess RuleKeeper's usability, we conducted a user study focused on answering two main questions: Q1) How much effort do developers take to specify a GDPR manifest based on a privacy policy written in English? Q2) How hard is it for them to debug inconsistencies between GDPR manifest and application code in the static analysis phase?

Methodology: As in prior usability studies [53–55], our subject group consists of ten participants. Since CS students are considered acceptable substitutes for developers [56, 57], we recruited CS students from our university. To minimize bias, candidates had no GDPR background, no prior knowledge of the test applications, and no specialized training in security. Five were MSc students with general CS education; five were Ph.D. students in distributed systems. We collected no personally identifiable information and did not pay them monetarily. Then, we asked them to perform two tasks.

1) *Specification task*: To study Q1, we asked participants to write the GDPR manifest from scratch for a realistic, non-trivial policy. We selected the LEB use case because it has a non-trivial data model without being an overly complex and time-consuming application to analyze. LEB's relative simplicity helps to keep participants focused on RuleKeeper-related tasks. We gave them the LEB's privacy policy written in natural language and the application source code and asked them to write a GDPR manifest that reflected both the application and the privacy policy requirements. Participants were encouraged to run RuleKeeper to check for inconsistencies.
2) *Debugging task*: As for Q2, the goal was to solve an inconsistency detected by the static analysis. We introduced an "incompatible purpose" violation in the LEB's source code reproducing Webus's bug described in *Example 1* of Section 2.2. We changed one of LEB's operations to access data that was not meant to be used for the operation's original purposes. We asked participants to detect and fix this inconsistency using RuleKeeper.

We provided the participants with i) a guide introducing RuleKeeper using Webus as an example (similar to Section 3.1) and ii) a `Vagrantfile` to set up a VM containing the static analysis tool and the use cases' source code (available in [29]). We gave them 45 minutes of training time to read the guide and study the source code of the LEB application.

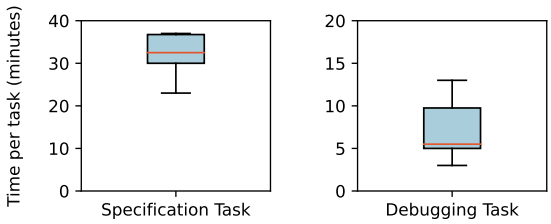

Figure 11: Time for participants to perform the tasks. The box extends from the 1st to the 3rd quartile; the median line is shown.

We measured: the total time each one took to complete a task (*c1*), number of correctly specified DSL statements (*c2*), and ability to detect and solve the inconsistency using RuleKeeper (*c3*). In the end, we asked for general feedback.

Findings: Concerning criteria *c1*, all ten participants completed both tasks within reasonable times (see Figure 11). For the specification task, they took on average 34.1 minutes, and for the debugging task, as few as 7.1 minutes to solve the inconsistency. As for manifest correctness (*c2*), one participant failed to specify a personal data item, resulting in 4 missing DSL statements out of the 66 expected ones. All the other nine manifests were fully correct. Regarding *c3*, all the participants managed to fix the bug. Their solution was to update the manifest's `collected-for` rule to include the additional accessed personal data, thus offering end-users complete transparency. When interviewed after the study, they all highlighted the simplicity of RuleKeeper's DSL. Most participants (60%) said having spent more time writing down the manifest's statements than reasoning about the logic, which suggests that a visual programming interface may help speed up this task. Their feedback was very positive, underlining RuleKeeper's usefulness as a debugging tool. For instance, seven participants reported having typos on their first try but were able to detect them quickly on account of the output of our static analysis tool.

8. Security Analysis

Detection of compliance bugs: To evaluate if RuleKeeper can mitigate the GDPR compliance threats of Section 2.2, we injected similar bugs in Webus and replicated them for the other use cases. For examples 1, 2, and 3, RuleKeeper blocks all unlawful queries. Surprisingly, in Habitica, we detected additional queries being blocked by RuleKeeper. This happened because Habitica was processing unnecessary user-related data for several operations, therefore violating the data minimization principle. We then rectified Habitica's code by removing accesses to that data from `GET /user` and `POST /user/auth/local/login` and confirmed that it remained as functional as before. This finding strengthens RuleKeeper's value in detecting existing compliance bugs.

Static analysis: Table 5 shows several evaluation metrics for the static analysis running across our use cases except for Habitica; in Habitica, compliance bugs can be detected at runtime only as it merely defines the Express.js' `Router` at the time of execution. We collected: CRG size, execution

Application	CRG Size		Execution Time (s)			Accuracy
	Nodes	Edges	CRG	DPG Queries (N,C)		EM-pairs
LEB (257 LoC)	1047	1710	0.201	27.814	21.213	10/11
Amazona (570 LoC)	2238	4508	0.357	41.154	34.520	16/16
Blog (1075 LoC)	4189	8987	0.637	589.669	540.623	31/34

Table 5: Static analysis engine metrics.

times, and accuracy metrics. The CRG size increases almost linearly on the application LoC size. We measured the execution times of two distinct stages: i) CRG generation, and ii) CRG querying to extract the DPG. For the second stage, we gauge execution times with (C) and without (N) the Neo4j cache. For larger graphs, the queries' execution time grows to the order of minutes, but it does not affect the application execution time as static analysis runs offline and only occasionally when changes occur in the application code or manifest. Regarding accuracy, our static analysis correctly identified 57 out of 61 endpoint-model (EM) pairs, where EM pairs correspond to the queries executed in the context of an endpoint. The missed EM pairs occur as a limitation of static analysis (see Appendix B for details).

GDPR compliance analysis: While RuleKeeper can verify purpose limitation, data minimization, and lawfulness of processing properties, it alone cannot guarantee full GDPR compliance as this regulation is much broader (see Appendix C). Nevertheless, RuleKeeper partially improves transparency, as sticky banners are automatically generated from the GDPR manifest albeit missing information such as data retention policies. Security of processing is improved by RuleKeeper's access control mechanisms. With future extensions, RuleKeeper can also implement storage limitation and accuracy preservation policies. Currently, it does not prevent GDPR violations caused by client-side scripts, as it is out of the scope of our work. However, this issue could be tackled with similar techniques to PoliCheck [58]. SDP [59] and user shards [60] can help cover complementary GDPR requirements to RuleKeeper's, namely storage limitation and accountability, and data subjects' rights, respectively.

Threat analysis: We discuss two attacks attempting to subvert RuleKeeper's policy enforcement mechanisms:

- *Middleware tampering*: RuleKeeper's HTTP hook is saved by Express.js in an array that contains Express's middleware stack; the DB hook is also stored by Mongoose in an array containing the Mongoose plugins. If a remote attacker can gain access to these variables, he can disable RuleKeeper's hooks by simply modifying the arrays. To prevent this, RuleKeeper disallows further modifications to the arrays by changing their properties [61].
- *Cookie theft attacks*: In an alternative attack [62], attackers may attempt to hijack the cookies used by RuleKeeper, which could leak private user information and be used to impersonate users. To mitigate these attacks, RuleKeeper uses session cookies set with the `Secure` and `HttpOnly` flags [63], which ensures the cookie is only sent in an encrypted request and makes the cookie inaccessible to unintended client-side APIs, respectively. To prevent session fixation attacks [64], RuleKeeper regenerates and resends the session cookies everytime a user authenticates.

9. Related Work

General studies on privacy and GDPR. Previous work [19, 65–67] examined concerns of end-users and developers on the security and privacy of applications. Recent studies [59, 68–70] focus on the tension between GDPR requirements and practices of modern computing systems. Shastri et al. [68] discuss current avoidable practices that collide with the GDPR. Shah et al. [70] illustrate the challenges of retrofitting existing storage systems into compliance. Experimental studies [18, 21–23, 71–73] investigate best practices and limitations in GDPR compliance. There are also relevant DSLs for privacy systems [74–76], but provide abstractions that do not easily map to our problem domain.

Policy enforcement for database-backed applications. Some systems track information flows by modifying the compiler and runtime for a managed language [77–80] or across the application/database boundary [15, 81–83]. Other IFC approaches [16, 84] enforce data-dependent security policies at compile-time only, being unable to perform dynamic access controls. Riverbed [13] is a practical IFC system enforcing user-defined policies that restrict the data online services can share with third parties. However, Riverbed's usage scenario is different from ours, lacking support for global policies and requiring the deployment of trusted hardware. The database community has explored fine-grained access-control techniques based on query rewriting [11, 12, 85, 86]. However, their fundamental limitation is the lack of application-specific context which precludes the ability to manage user consent preferences.

Privacy policy generation and auditing. Compliance analysis systems have been proposed to i) identify discrepancies between privacy policies and actual code functionality [58, 87, 88] and ii) aid developers write privacy policies for applications [17, 89]. Some systems are specific for mobile applications [17, 58, 87–90]. PrivGuard [14] employs static analysis and trusted execution environments for validating compliance of Python programs with regulations such as the GDPR. However, this system was designed for different goals than ours, proposing a specialized solution for data analytics and machine learning workloads that is unsuitable for typical web development.

10. Conclusions

This work presents RuleKeeper, a GDPR-aware policy compliance system for full-stack web development frameworks. It includes a policy specification language and employs static and dynamic analysis to enforce policy compliance, and was implemented for the MERN web stack. RuleKeeper can prevent various GDPR compliance violations, while registering acceptable performance overheads. As for future work, we intend to i) extend the verified GDPR guidelines, ii) improve the accuracy of static analysis, iii) optimize the performance, e.g., through query rewriting, and iv) incorporate a complementary accountability infrastructure for tracking both the offline phase (e.g., cryptographic code signatures) and the online phase (e.g., security logs).

Acknowledgments. We thank our shepherd and the anonymous reviewers for their comments and insightful feedback. We also thank all the participants of the usability study. This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) via the 2021.06134.BD and SFRH/BD/146698/2019 grants, and the UIDB/50021/2020, PTDC/CCI-COM/32378/2017 (INFOCOS), and CMU/TIC/0053/2021 (DIVINA) projects.

References

[1] The European Parliament and the Council of the European Union, "Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)," *Official Journal of the European Union*, vol. L 119, May 2016.

[2] Commission Nationale de l'Informatique et des Libertés, "The cnil's restricted committee imposes a financial penalty of 50 million euros against google llc," 2022, retrieved January 11, 2022 from https://www.cnil.fr/en/cnils-restricted-committee-imposes-financial-penalty-50-million/-euros-against-google-llc.

[3] European Data Protection Board, "Norwegian dpa imposes fine against grindr llc," 2022, retrieved January 11, 2022 from https://edpb.europa.eu/news/national-news/2021/norwegian-dpa-imposes-fine-against-grindr-llc_en.

[4] United States Securities and Exchange Commission, "Form 10-q - for the quarterly period ended june 30, 2021," 2021, retrieved January 11, 2022 from https://www.sec.gov/Archives/edgar/data/0001018724/000101872421000020/amzn-20210630.htm.

[5] European Data Protection Board, "Administrative criminal proceedings of the austrian data protection authority against Österreichische post ag," 2022, retrieved January 11, 2022 from https://edpb.europa.eu/news/national-news/2019/administrative-criminal-proceedings-austrian-data-protection-authority_en.

[6] MongoDB, "Mern stack explained," 2021, retrieved January 13, 2022 from https://www.mongodb.com/mern-stack.

[7] ReactJS, "A javascript library for building user interfaces," 2022, retrieved January 11, 2022 from https://reactjs.org/.

[8] Express.js, "Fast, unopinionated, minimalist web framework for node.js," 2021, retrieved January 11, 2022 from https://expressjs.com.

[9] Node.js, "Javascript runtime built on chrome's v8 javascript engine," 2022, retrieved January 11, 2022 from https://nodejs.org/.

[10] MongoDB, "Mongodb: the application data platform," 2021, retrieved January 11, 2022 from https://www.mongodb.com.

[11] A. Bichhawat, M. Fredrikson, J. Yang, and A. Trehan, "Contextual and granular policy enforcement in database-backed applications," in *Proc. of AsiaCCS*, 2020.

[12] A. Mehta, E. Elnikety, K. Harvey, D. Garg, and P. Druschel, "Qapla: Policy compliance for database-backed systems," in *Proc. of USENIX Security*, 2017.

[13] F. Wang, R. Ko, and J. Mickens, "Riverbed: Enforcing user-defined privacy constraints in distributed web services," in *Proc. of NSDI*, 2019.

[14] L. Wang, U. Khan, J. Near, J. S. Qi Pang, N. Somani, P. Gao, A. Low, and D. Song, "PrivGuard: Privacy regulation compliance made easier," in *Proc. of USENIX Security*, 2022.

[15] M. Guarnieri, M. Balliu, D. Schoepe, D. Basin, and A. Sabelfeld, "Information-flow control for database-backed applications," in *Proc. of EuroS&P*, 2019.

[16] N. Lehmann, R. Kunkel, J. Brown, J. Yang, N. Vazou, N. Polikarpova, D. Stefan, and R. Jhala, "Storm: Refinement types for secure web applications," in *Proc. of OSDI*, 2021.

[17] S. Zimmeck, R. Goldstein, and D. Baraka, "Privacyflash pro: automating privacy policy generation for mobile apps," in *Proc. of NDSS*, 2021.

[18] C. Matte, N. Bielova, and C. Santos, "Do cookie banners respect my choice? measuring legal compliance of banners from iab europe's transparency and consent framework," *Proc. of S&P*, 2020.

[19] R. Khandelwal, T. Linden, H. Harkous, and K. Fawaz, "PriSEC: A privacy settings enforcement controller," in *Proc. of USENIX Security*, 2021.

[20] D. Bollinger, K. Kubicek, C. Cotrini, and D. Basin, "Automating cookie consent and GDPR violation detection," in *Proc. of USENIX Security*, 2022.

[21] M. Nouwens, I. Liccardi, M. Veale, D. Karger, and L. Kagal, *Dark Patterns after the GDPR: Scraping Consent Pop-Ups and Demonstrating Their Influence*, 2020.

[22] C. Utz, M. Degeling, S. Fahl, F. Schaub, and T. Holz, "(un)informed consent: Studying gdpr consent notices in the field," in *Proc. of CCS*, 2019.

[23] T. T. Nguyen, M. Backes, N. Marnau, and B. Stock, "Share first, ask later (or never?) studying violations of GDPR's explicit consent in android apps," in *Proc. of USENIX Security*, 2021.

[24] Cookie Consent, "Cookie consent notice banner to comply with gdpr and eprivacy directive," 2022, retrieved August 18, 2022 from https://www.cookieconsent.com/.

[25] react-cookie-consent, "A small, simple and customizable cookie consent bar for use in react applications," 2022, retrieved August 18, 2022 from https://www.npmjs.com/package/react-cookie-consent.

[26] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. of S&P*, 2014.

[27] S. Khodayari and G. Pellegrino, "JAW: Studying client-side CSRF with hybrid property graphs and declarative traversals," in *Proc. of USENIX Security*, 2021.

[28] S. Li, M. Kang, J. Hou, and Y. Cao, "Mining node.js vulnerabilities via object dependence graph and query," in *Proc. of USENIX Security*, 2022.

[29] RuleKeeper supplementary material, 2022, https://github.com/rulekeeper/rulekeeper.

[30] A. Fass, M. Backes, and B. Stock, "JSTAP: A static pre-filter for malicious javascript detection," in *Proc. of ACSAC*, 2019.

[31] C.-A. Staicu, D. Schoepe, M. Balliu, M. Pradel, and A. Sabelfeld, "An empirical study of information flows in real-world javascript," in *Proc. of PLAS*, 2019.

[32] A. Nagpal, R. Dasgupta, and B. Ganesan, "Fine grained classification of personal data entities with language models," in *Proc. of CODS-COMAD*, 2022.

[33] Mongoose, "Elegant mongodb object modeling for node.js," 2022, retrieved January 13, 2022 from https://mongoosejs.com/.

[34] Cloud Native Computing Foundation, "Open Policy Agent (OPA)," 2022, retrieved January 11, 2022 from https://www.openpolicyagent.org/.

[35] ——, "Rego Policy Language," 2022, retrieved January 13, 2022 from https://www.openpolicyagent.org/docs/latest/policy-language.

[36] WebAssembly, 2021, retrieved January 11, 2022 from https://webassembly.org/.

[37] Socket.IO, "Bidirectional and low-latency communication for every platform," 2022, retrieved January 13, 2022 from https://socket.io/.

[38] Esprima, "Ecmascript parsing infrastructure for multipurpose analysis," 2021, retrieved January 11, 2022 from https://esprima.org/index.html.

[39] Neo4j, "Neo4j graph data platform," 2022, retrieved January 13, 2022 from https://neo4j.com/.

[40] Stack Overflow, "Stack overflow developer survey results 2021," 2022, retrieved November 17, 2022 from https://insights.stackoverflow.com/survey/2021.

[41] GitHub, "The 2021 state of the octoverse top languages," 2021, retrieved November 17, 2022 from https://octoverse.github.com/.

[42] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow, "Deemon: Detecting csrf with dynamic analysis and property graphs," in *Proc. of CCS*, 2017.

[43] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "NAVEX: Precise and scalable exploit generation for dynamic web applications," in *Proc. of USENIX Security*, 2018.

[44] Análises Clínicas LEB - Laboratórios Elisabeth Barreto, 2022, retrieved January 14, 2022 from https://www.leb-analises.com/.

[45] APAC - Associação Portuguesa de Analistas Clínicos, 2022, retrieved April 18, 2022 from https://www.apaclinicos.pt/.

[46] HabitRPG, "habitica - release v4.189.0," 2021, retrieved January 14, 2022 from https://github.com/HabitRPG/habitica/releases/tag/v4.189.0.

[47] Google Play, "Habitica: Gamify your tasks, by habitrpg, inc." 2021, retrieved January 14, 2022 from https://play.google.com/store/apps/details?id=com.habitrpg.android.habitica.

[48] Basir, "Amazona - build ecommerce website like amazon," 2020, retrieved April 13, 2022 from https://github.com/basir/node-react-ecommerce.

[49] gothinkster, "Blog - realworld example app," 2018, retrieved April 13, 2022 from https://github.com/gothinkster/node-express-realworld-example-app.

[50] Passport, "npm - passport," 2021, retrieved April 13, 2022 from https://www.npmjs.com/package/passport.

[51] wrk, "wrk - a http benchmarking tool," 2021, retrieved January 13, 2022 from https://github.com/wg/wrk.

[52] linux.die.net, "dstat(1) - linux man page," 2022, retrieved January 14, 2022 from https://linux.die.net/man/1/dstat.

[53] K. Caine, "Local standards for sample size at chi," in *Proc. of CHI*, 2016.

[54] W. Hwang and G. Salvendy, "Number of people required for usability evaluation: The 10±2 rule," *Communications of the ACM*, p. 130–133, May 2010.

[55] Y. Beugin, Q. Burke, B. Hoak, R. Sheatsley, E. Pauley, G. Tan, S. R. Hussain, and P. McDaniel, "Building a Privacy-Preserving Smart Camera System," in *Proc. of PETS*, 2022.

[56] Y. Acar, C. Stransky, D. Wermke, M. L. Mazurek, and S. Fahl, "Security developer studies with GitHub users: Exploring a convenience sample," in *Proc. of SOUPS*, 2017.

[57] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?" in *IEEE International Conference on Software Engineering*.

[58] B. Andow, S. Y. Mahmud, J. Whitaker, W. Enck, B. Reaves, K. Singh, and S. Egelman, "Actions speak louder than words: Entity-Sensitive privacy policy and data flow analysis with PoliCheck," in *Proc. of USENIX Security*, 2020.

[59] Z. István, S. Ponnapalli, and V. Chidambaram, "Software-defined data protection: Low overhead policy compliance at the storage layer is within reach!" *Proc. of VLDB Endowment*, vol. 14, no. 7, p. 1167–1174, 2021.

[60] M. Schwarzkopf, E. Kohler, M. F. Kaashoek, and R. T. Morris, "Position: Gdpr compliance by construction," in *Poly/DMAH@VLDB*, 2019.

[61] mdn web docs, "Object.defineproperties() - javascript — mdn," 2021, retrieved April 14, 2022 from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperties.

[62] S. Sivakorn, I. Polakis, and A. D. Keromytis, "The cracked cookie jar: Http cookie hijacking and the exposure of private information," in *Proc. of S&P*, 2016.

[63] mdn web docs, "Using http cookies - http — mdn," 2022, retrieved April 14, 2022 from https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies.

[64] OWASP, "Session fixation software attack — owasp foundation," 2022, retrieved April 14, 2022 from https://owasp.org/www-community/attacks/Session_fixation.

[65] L. Zhang-Kennedy and S. Chiasson, ""whether it's moral is a whole other story": Consumer perspectives on privacy regulations and corporate data practices," in *Proc. of SOUPS*, 2021, pp. 197–216.

[66] M. Tahaei, K. Vaniea, and N. Saphra, *Understanding Privacy-Related Questions on Stack Overflow*, 2020.

[67] R. Balebako, A. Marsh, J. Lin, J. Hong, and L. F. Cranor, "The privacy and security behaviors of smartphone app developers," in *Proc. of USEC*, 2014.

[68] S. Shastri, M. Wasserman, and V. Chidambaram, "The seven sins of personal-data processing systems under gdpr," in *Proc. of HotCloud*, 2019.

[69] J. Mohan, M. Wasserman, and V. Chidambaram, "Analyzing GDPR compliance through the lens of privacy policy," pp. 82–95, 2019.

[70] A. Shah, V. Banakar, S. Shastri, M. Wasserman, and V. Chidambaram, "Analyzing the impact of gdpr on storage systems," in *Proc. of HotStorage*, 2019.

[71] M. Degeling, C. Utz, C. Lentzsch, H. Hosseini, F. Schaub, and T. Holz, "We value your privacy ... now take some cookies: Measuring the gdpr's impact on web privacy," in *Proc. of NDSS*, 2019.

[72] E. Papadogiannakis, P. Papadopoulos, N. Kourtellis, and E. P. Markatos, *User Tracking in the Post-Cookie Era: How Websites Bypass GDPR Consent to Track Users*, 2021.

[73] I. Sanchez-Rola, M. Dell'Amico, P. Kotzias, D. Balzarotti, L. Bilge, P.-A. Vervier, and I. Santos, "Can i opt out yet? gdpr and the global illusion of cookie control," in *Proc. of AsiaCCS*, 2019.

[74] S. Sen, S. Guha, A. Datta, S. Rajamani, J. Tsai, and J. Wing, "Bootstrapping privacy compliance in big data systems," in *Proc. of S&P*, 2014.

[75] F. Karami, D. Basin, and E. B. Johnsen, "DPL: A Language for GDPR Enforcement," in *Proc. of CSF*, 2022.

[76] E. Bagdasaryan, G. Berlstein, J. Waterman, E. Birrell, N. Foster, F. B. Schneider, and D. Estrin, "Ancile: Enhancing privacy for ubiquitous computing with use-based privacy," in *Proc. of WPES*, 2019.

[77] S. Chong, K. Vikram, and A. C.Myers, "Sif: Enforcing confidentiality and integrity in web applications," in *Proc. of USENIX Security*, 2007.

[78] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers, "Fabric: A platform for secure distributed computation and storage," in *Proc. of SOSP*, 2009.

[79] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Improving application security with data flow assertions," in *Proc. of SOSP*, 2009.

[80] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo, "Hails: Protecting data privacy in untrusted web applications," in *Proc. of OSDI*, 2012.

[81] M. Balliu, B. Liebe, D. Schoepe, and A. Sabelfeld, "Jslinq: Building secure applications across tiers," in *Proc. of CODASPY*, 2016.

[82] B. J. Corcoran, N. Swamy, and M. W. Hicks, "Cross-tier, label-based security enforcement for web applications," in *Proc. of COMAD*, 2009.

[83] D. Schoepe, D. Hedin, and A. Sabelfeld, "Selinq: Tracking information across application-database boundaries," in *Proc. of ICFP*, 2014.

[84] A. Chlipala, "Static checking of dynamically-varying security policies in database-backed applications," in *Proc. of OSDI*, 2010.

[85] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy, "Extending query rewriting techniques for fine-grained access control," in *Proc. of COMAD*, 2004.

[86] M. Guarnieri, S. Marinovic, and D. Basin, "Strong and provably secure database access control," in *Proc. of EuroS&P*, 2016.

[87] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proc. of ICSE*, 2014.

[88] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," *Proc. of CCS*, 2014.

[89] H. Wang, Y. Li, Y. Guo, Y. Agarwal, and J. I. Hong, "Understanding the purpose of permission use in mobile apps," *ACM Transactions on Information Systems*, vol. 35, no. 4, 2017.

[90] D. Bui, Y. Yao, K. G. Shin, J.-M. Choi, and J. Shin, "Consistency analysis of data-usage purposes in mobile apps," in *Proc. of CCS*, 2021.

[91] S. Shastri, V. Banakar, M. Wasserman, A. Kumar, and V. Chidambaram, "Understanding and benchmarking the impact of gdpr on database systems," *Proc. of VLDB Endowment*, vol. 13, no. 7, p. 1064–1077, 2020.

[92] E. Arfelt, D. Basin, and S. Debois, "Monitoring the GDPR," in *Computer Security – ESORICS 2019*. Springer-Verlag, p. 681–699.

Appendix A.
Case Studies

This section presents complementary information on the case studies presented in Section 5. Pertaining to the LEB use case, we analyzed its internal processes that manage personal data: *pre-analytic* is responsible for patient registration and specimen processing, *analytic* for specimen analysis and validation, *post-analytic* for preparing and emitting the analysis results, and *human resources* for hiring and training new employees. APAC [45] assigned the purpose "clinical analysis" to the clinical processes, and "human resources" to the human resources process. Given the extent of these internal processes and the fact that most of them process the same data, our prototype focuses on the *pre-analytic* process, responsible for patient registration and specimen processing. Figure 12 details LEB's pre-analytic process, where decision blocks are represented in yellow, process blocks are represented in blue and terminal blocks in orange. As we wanted to extend this prototype with RuleKeeper, we focused on actions that involve the patients' personal data processing. Table 6 describes the seven controllers that we implemented as part of LEB's intranet service for supporting the pre-analytic process of the laboratory.

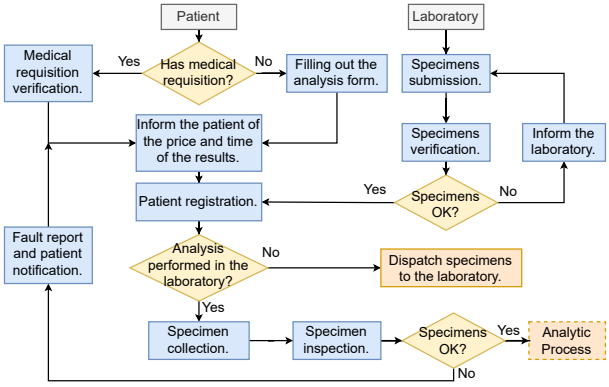

Figure 12: Flowchart of LEB's pre-analytic process.

Controller	Description
Register patient	Receptionists register new patients in the system.
Access patient	Receptionists fetch patients' data. Patients fetch their own data.
Access several patients	Receptionists fetch data from patients to query some parameters, such as checking appointments.
Update patient	Receptionists update patient's data. A patient can update some aspects of its own data.
Register user	System administrators register users in the system.
Delete user	System administrators delete users in the system.
Update user	System administrators update users in the system.

Table 6: Controllers in the LEB prototype intranet service.

Given that our other case studies are based on legacy applications, we studied their source code to detect their operations. For the Amazona and Blog use case, we considered all the operations. Habitica's API comprises over 200 routes, so, we considered the operations that we considered most relevant, grouped by category, as described in Table 7.

Category	Operations
Users	Register user, get user's profile, get user's rewards
Tasks	Create a new task , get user's tasks, score a task
Groups	Create a new group, invite member to group, join group
Challenges	Create a new challenge, join challenge, select winner
Quests	Invite users to a quest, accept a pending quest
Halls	Get all heroes (contributors), update hero (any user)
News	Get latest news, create a new news post
Coupons	Get coupons, redeem a coupon

Table 7: Habitica's controllers.

```
const express = require('express');
const mongoose = require('mongoose');
const flatten = require('flat');

const patientRouter = express.Router();
const Patient = mongoose.model('patients');
// (...)

/* Update patient data. */
patientRouter.post('/patients/:patientId', function (req, res) {
  const { patientId } = req.params;
  let data = req.body; // data contains the colums to return

  if (data) data = flatten(data);
  const patient = { citizencard: patientId };
  Patient.findOneAndUpdate(patient, data, ...);
});
```

Figure 13: Code snippet of LEB application for which static analysis fails to detect columns being updated.

Appendix B.
Illustration of Static Analysis Limitations

In Section 6, we show that our static analysis engine fails to correctly detect one endpoint-model (EM) pair from the LEB use case and three EMs for the Blog use case. At their root cause, lies the same limitation. Listing 13 shows the code snippet relative to the missed LEB EM pair.

Our static analysis detects the registration of the endpoint `POST /patients/:patientId` and the use of the `findOneAndUpdate` function being called on the `Patient` model, which corresponds to the `patients` table in the database. However, it cannot detect the specific parameters that are being updated, which correspond to the properties of the `data` object. The properties of the `data` object can only be known at runtime, as they depend on the user input sent via the body of the request object (`req`). From our experience, sending input objects directly to a database statement, such as `create` or `findOneAndUpdate`, is very uncommon and a security bad practice. Typically, EM pairs resemble the code shown in Listing 14, where a new object is created (`user`), whose properties depend on data from the input (`req.body`).

In this example, the static analysis approach can detect the properties of the `user` object, given as a parameter of the `create` Mongoose function, and the structure of this object does not change for different inputs. To handle the lack of knowledge of the object properties, we consider that all data fields (as described in the data model) are being accessed, which can lead to false positives – only known at runtime. This strengthens the need for a second line of defense for preventing GDPR violations, at runtime.

No.	GDPR article	Property	Support
5.1 b)	PURPOSE LIMITATION	Data must be collected and used for specific purposes.	●
5.1 c)	DATA MINIMIZATION	Data must be necessary to the purposes for which it is processed.	●
5.1 e)	STORAGE LIMITATION	Data should not be stored beyond its purpose.	○
5.1 a); 6	LAWFULNESS OF PROCESSING	Each purpose must have a valid lawful reason to process the personal data.	●
5.1 a); 12; 13; 14	TRANSPARENCY & INFORMATION TO BE PROVIDED	The data subject should be made aware that its personal data is being collected and to what extent it is or will be processed.	◐
5.1 d)	ACCURACY PRESERVATION	Data must be accurate and kept up to date.	○
5.1 d); 30; 33; 34	ACCOUNTABILITY	Controller must be able to demonstrate compliance.	○
32	SECURITY OF PROCESSING	The controller must implement appropriate data security measures.	◐

Table 8: Key GDPR articles. ● indicates that RuleKeeper supports full compliance, ○ that it does not.

```
const express = require('express');
const mongoose = require('mongoose');

const userRouter = express.Router();
const User = mongoose.model('users');
// (...)

/* Create user. */
userRouter.post('/users/register', function (req, res) {
  const user = {
    username: req.body.username,
    password: hash(req.body.password)
  };
  User.create(user, ...);
});
```

Figure 14: Code snippet of a typical EM pair example.

Appendix C. Summary of GDPR Guidelines

Some GDPR regulation principles lie in grey areas, and are not natively supported in information systems, raising a lot of challenges for organizations who want to be compliant with the regulation. This challenge has been addressed by several studies [18, 68, 69, 91, 92], which discuss the implications of GDPR compliance in information systems Table 8 summarizes the relevant key requirements that we have identified by extensively analyzing the articles of the GDPR and the existing studies. For each property, the table indicates the GDPR articles that support it, a brief summary of the requirement, and RuleKeeper's support for GDPR-compliance, where ● indicates that RuleKeeper supports full compliance and ○ indicates that it does not. As originally intended, RuleKeeper fully supports purpose limitation, data minimization, and lawfulness of processing properties.

Appendix D. RuleKeeper's DSL specification

Table 9 presents the specification of RuleKeeper's domain-specific language. This DSL is used by web developers to write the GDPR manifest that specifies the personal data types and purposes for which the web application can process the data. An example of an instance of this DSL for the Webus use case is presented in Section 3.1.

D.1. Illustration of an inconsistency between the GDPR manifest and the application

RuleKeeper's static code analysis pipeline helps to look for inconsistencies between the application code and the GDPR manifest, as described in Section 3.2. In this section, we present an example of inconsistency for the Webus use case, where data is processed for incompatible purposes, similar to the one presented to the participants in the debugging task of the usability study (see Section 7). Listing 15 replicates the code snippet of the "subscribe to newsletter" operation presented in Section 2.2. This operation

Statement	Rule
DATA-ITEMS-DECL	**DATA-ITEMS**: list(<string>)
OPERATIONS-DECL	**OPERATIONS**: list(<string>)
PERSONAL-DATA-DECL	**PERSONAL-DATA**: list(<string>)
PURPOSES-DECL	**PURPOSES**: list(<string>)
ROLES-DECL	**ROLES**: list(<string>)
PREAMBLE	<data-items-decl> <operations-decl> <personal-data-decl> <purposes-decl> <roles-decl>
DATA-COLLECTION-CLAUSE	list(<string>) **ARE COLLECTED FOR** <string> **purposes**
DATA-COLLECTION-DECL	**DATA-COLLECTION**: list(<data-collection-clause>)
LAWFULNESS-CLAUSE	**PURPOSE** <string> **HAS LAWFULNESS BASE** <string>
LAWFULNESS-BASE-DECL	**LAWFULNESS-BASE**: list(<lawfulness-clause>)
EXECUTED-FOR-CLAUSE	list(<string>) **ARE EXECUTED FOR** <string> **purposes**
EXECUTED-FOR-DECL	**EXECUTED-FOR**: list(<executed-for-clause>)
DATA-MAPPING-CLAUSE	<string> **IS IN COLUMN** <string> **OF TABLE** <string>
DATA-MAPPING-DECL	**DATA-MAPPING**: list(<data-mapping-clause>)
OPERATION-MAPPING-CLAUSE	<string> **IS MAPPED TO ENDPOINT** <string>
OPERATION-MAPPING-DECL	**OPERATION-MAPPING**: list(<operation-mapping-clause>)
DATA-OWNERSHIP-CLAUSE	**OWNER IN TABLE** <string> **IS IN COLUMN** <string>
DATA-OWNERSHIP-DECL	**DATA-OWNERSHIP**: list(<data-ownership-clause>)
AUTHORIZED-ROLES-CLAUSE	**ROLE** <string> **IS AUTHORIZED TO** list(<string>)
AUTHORIZED-ROLES-DECL	**AUTHORIZED-ROLES**: list(<authorized-roles-clause>)
PROGRAM	<preamble> <data-collection-decl> <lawfulness-base-decl> <executed-for-decl> <data-mapping-decl> <operation-mapping-decl> <data-ownership-decl> <authorized-roles-decl>

Table 9: RuleKeeper's DSL specification.

```
function subscribe(req, res) {
 const { e_mail } = req.body;

 connection.query(`INSERT INTO newsletters (e_mail) VALUES
 ↪  ('${e_mail}')`, (err) => {

  connection.query("SELECT * FROM tickets WHERE e_mail = '${e_mail}'
  ↪  AND year(date)>=2021", (err, tickets) => {
    if (tickets.length >= 10) sendPromoCode(e_mail);
    res.sendStatus(200); });
  });
}
```

Figure 15: Code snippet of the Webus' "subscribe to newsletter" operation.

```
DATA-ITEMS: ticket buyer email, ticket buyer credit card, email
OPERATIONS: buy ticket, subscribe to newsletter
PERSONAL-DATA: ticket buyer email, ticket buyer credit card, email
PURPOSES: ticket management, marketing
DATA-COLLECTION:
  ticket buyer email, ticket buyer credit card ARE COLLECTED FOR
  ↪  ticket management purposes
  email IS COLLECTED FOR marketing purposes
EXECUTED-FOR:
  buy ticket, see purchase history ARE EXECUTED FOR ticket
  ↪  management purposes
  subscribe to newsletter IS EXECUTED FOR marketing
DATA-MAPPING:
  ticket buyer email IS IN COLUMN name OF TABLE tickets.
  ticket buyer credit card IS IN COLUMN credit_card OF TABLE
  ↪  tickets.
  email IS IN COLUMN e_mail OF TABLE newsletter.
OPERATION-MAPPING:
  buy ticket IS MAPPED TO ENDPOINT POST /buy_ticket.
  subscribe to newsletter IS MAPPED TO ENDPOINT POST /subscribe.
```

Figure 16: Simplified version of RuleKeeper's policy for Webus.

comprises two steps: (i) insert the user's email in the *Newsletter* table, and (ii) send a promotional code to frequent travelers ($\geqslant$ 10 trips in 2021). To check if a user is a frequent traveler, step *ii* accesses the *Ticket* table, which contains the user's email and credit card. Figure 16 shows a simplified version of the Webus policy presented in Section 3.1.

Data processed for incompatible purposes: The `EXECUTED-FOR` rule specifies that the "subscribe to newsletter" operation is executed for marketing purposes, which is only allowed to process the users' email, as per defined in the `DATA-COLLECTION` rule. However, step *ii* accesses the users' email and credit card, which are only allowed to be processed for ticket management purposes. In this case, RuleKeeper's static analysis will detect a purpose limitation inconsistency, where the application is processing more personal data for a given purpose than what is advertised by the GDPR manifest.