Towards Application-Layer Purpose-Based Access Control

— Preprint ACM SAC 2020 —

Frank Pallas
TU Berlin
Information Systems Engineering
Research Group
Berlin, Germany
fp@ise.tu-berlin.de

Max-R. Ulbricht
TU Berlin
Information Systems Engineering
Research Group
Berlin, Germany
mu@ise.tu-berlin.de

Stefan Tai
TU Berlin
Information Systems Engineering
Research Group
Berlin, Germany
st@ise.tu-berlin.de

Thomas Peikert
TU Berlin
thomas.peikert@campus.tu-berlin.de

Marcel Reppenhagen
TU Berlin
marcel.reppenhagen@campus.tu-berlin.de

Daniel Wenzel
TU Berlin
daniel.wenzel@campus.tu-berlin.de

Paul Wille
TU Berlin
paul.wille@campus.tu-berlin.de

Karl Wolf
TU Berlin
karl.wolf@campus.tu-berlin.de

Preprint, to appear in: The 35th ACM/SIGAPP Symposium On Applied Computing (ACM SAC 2020), Brno, Czech Republic, March 30-April 3, 2020.
Final version availible at:
https://doi.org/10.1145/3341105.3375764

ABSTRACT

In this paper, we propose an architecturally novel approach to implementing purpose-based access control in practice. Different from previous proposals, our approach resides on the application instead of the data(base) layer. This allows for significantly better integration with established architectures and practices of real-world application engineering and to achieve database independence.

To validate practical applicability, we provide two exemplary implementations and briefly assess the introduced overhead in matters of achievable throughputs. Results significantly depend on data and query type but basically suggest bearable overheads for realistic applications even though possible performance optimizations have not been implemented in our proofs-of-concept yet. Our approach thus proposes significantly better practical feasibility than previous ones and exhibits reasonable overheads. It therefore paves the way for purpose-based access control to be actually adopted in practice.

KEYWORDS

Privacy, data protection, purpose limitation, access control, PBAC, privacy by design, privacy engineering, web engineering

1 INTRODUCTION

Privacy by Design[1] (PbD) is one of the core concepts of modern privacy legislation, aiming at the effective implementation of privacy principles through concrete technologies and their design. For instance, the European General Data Protection Regulation (GDPR) requires data controllers to "implement appropriate technical and organisational measures [..] designed to implement data-protection principles [...] in an effective manner and to integrate the necessary safeguards into the processing" [10, Art. 25 (1)].

Noteworthily, this obligation refers to "data protection principles" in general. Even though (academic) discussions on Privacy by Design largely revolve around data minimization (anonymization, pseudonymization, etc.) and security, other principles therefore also need to be addressed properly.

Of the various established privacy principles and, in particular, those codified in Art. 5 of the GDPR, we herein concentrate on the principle of purpose limitation and the possibilities for addressing it technologically, "by Design". Basically, purpose limitation as a legislative obligation is codified in Art. 5 of the GDPR, stating that

> Personal data shall be [...] collected for specified, explicit and legitimate purposes and not further processed in a manner that is incompatible with those purposes [10, Art. 5 (1 b)]

Other examples vividly illustrating the particular relevance of purpose limitation throughout the regulatory regime of the GDPR include the limitation of individually provided consent to "one or more specific purposes" (Art. 6 (1)), the rigorous purpose restrictions for processing special categories of personal data laid down in Art. 9, or the obligations to inform data subjects about the purposes for which data is collected and processed given in Art. 13 ff. Keeping track of the purposes that certain pieces of personal data may be used for and ensuring these limitations to actually be complied with is thus a core obligation for any party collecting and processing personal data under the regime of the GDPR.

While the principle of purpose limitation has so far mostly been implemented through rather administrative procedures ("privacy by policy", according to Spiekermann and Cranor [31]), this approach proves increasingly inappropriate in the light of current and foreseeable givens. With omnipresent collection of personal data and its use for novel, initially unforeseen use-cases ("repurposing")

[1]Being well aware of the slightly different notions between "Privacy" and "Data Protection", we use these terms interchangeably herein.

across organizanional boundaries proposing significant user, business, and societal value, the administrative approach to purpose limitation is increasingly unfitting for practice. Preserving purpose limitation as a meaningful core principle of privacy under these conditions requires it to be reflected technically and, in particular, in a way that ties well into the current technological landscape.

Different approaches for technically implementing the principle of purpose-limitation have been proposed in the past under the term of "purpose-based access control (PBAC)" [e.g., 7, 8]. These approaches have, however, not made their way into practice, which can – at least partly – be attributed to two shortcomings of PBAC techniques proposed so far:

- Existing approaches are typically limited to one particular database system, thus limiting developers' flexibility and foreseeably imposing product lock-ins
- Existing approaches do not properly incorporate established concepts and architectural patterns from real-world application engineering, thus hindering actual adoption in practice.

In order to address these shortcomings and to pave the way for purpose-based access control to be used in practice, we therefore propose a novel, application-layer approach for realizing PBAC in a practical and economically feasible way. Our approach is at the same time database-agnostic and in line with common patterns of application engineering. In addition, it also allows to represent and automatically evaluate purposes in line with legal requirements (e.g., regarding different levels of specificity / categorizations), thus ensuring legal relevance and, in turn, actual practical benefit. In particular, we provide the following contributions:

- We identify a conceptual misfit between actual practices and patterns of application engineering and existing proposals for PBAC as root cause for PBAC's lack of adoption in practice
- We introduce a novel, development-friendly approach to implement PBAC on the application layer through native integration into Object-Relational Mappers
- We demonstrate practical viability of our approach through two validating implementations for two different development languages and stacks.

2 BACKGROUND & RELATED WORK

Our approach rests on two pillars of relevant background and context: On the one hand, this refers to the principle of purpose limitation, its role in privacy-related regulations, and previous approaches for reflecting it technically. On the other hand, we built upon a solid comprehension of modern application architectures and respective development practices. Both shall be sketched briefly.

2.1 Purpose Limitation & Access Control

The principle of purpose limitation as an instrument to prevent data subjects from being harmed by secret, hidden or otherwise unwanted processing of their personal data was deeply rooted in all kinds of privacy regulations – from the early beginnings of data protection laws in the 1970s to the European General Data Protection Regulation (GDPR) adopted in 2016.

Principle. Released in 1980, two of the basic principles within the OECD Guidelines on the Protection of Privacy and Transborder Flows of Personal Data [21][2] are the *Use Limitation Principle* and the *Purpose Specification Principle*. Together, these obligate the data processing entity to explicitly determine purposes for the collection and the processing of personal data and to restrict the further usage or disclosure of that data to purposes compatible with those initially specified.

The same pattern regarding the limitation of data collection and usage to previously specified purposes is also resembled in the 'Notice/Awareness' principle from the Federal Trade Commission's *Fair Information Practice Principles* [13] as well as in a combination of the 'Purpose legitimacy and specification' principle and the 'Use, retention and disclosure limitation' principle from the ISO/IEC standard 29100:2011 [15].

Besides these non-binding standards and guidelines, the principle of purpose limitation has also been anchored in binding legislations. The European *Data Protection Directive 95/46/EC* [9] of 1995 and its successor, the European General Data Protection Legislation [10], are among the most prominent of these. *Purpose Limitation* is now a legally backed principle every data processing entity that operates in Europe or provides services for European citizens has to respect.

Model. As a major step towards technically mediated enforcement of the purpose limitation principle, Byun et al. proposed a formalized model – Purpose-Based Access Control [5, 6]. In this model, purposes are defined as *reason(s) for the data collection and data access* (for the remainder of this paper, we subsume these activities as *processing*) and all possible processing purposes in a given domain are logically organized as a tree. It is thus possible to represent hierarchical relations between different purposes, facilitating the generalization and specialization of purposes. This allows for the specification of acceptable processing purposes in different levels of detail, depending on the particular needs within a given use-case.

In addition to the contribution of formalizing the purpose limitation principle in general, Byun et al. also proposed to split the concept of purpose into two distinct parts, namely *Allowed Purposes* and *Prohibited Purposes*. This splitting allows to specify comparably finegrained access policies with limited effort through, e.g., permitting access for a generalized category of purposes but at the same time excluding one more specific sub-purpose [7]. For instance, access may be granted for every research-related purpose except military ones. In particular, this allows to prevent unwanted data processing while at the same time providing greater utility of already existing data collections through re-purposing within the boundaries of given accepted purposes.

Implementation. An early approach to technically enforce purpose limitation is the concept of Hippocratic Databases [1]. The proposed architecture was designed to enhance Relational DataBase Management Systems (RDBMS) in a purpose-aware manner. For doing so, purpose-based access permissions – containing allowed processing purposes, recipients, and a retention period for every attribute – are stored in additional database tables and evaluated by the database system itself. Any data request must then refer to

[2]An updated version was published in 2013, see [22]

a desired processing purpose and is brought in line with given permissions by means of query modification so that only data actually matching the provided purpose is returned.

Despite the promising concept, real-world implementations ready for use in practice are still missing. Over the years, different prototypical implementations have been published, e.g. for IBM DB2 [18], PostgreSQL [17, 23], or the Microsoft SQL-Server [3]. An experimental evaluation of the original PBAC model was implemented with an Oracle 10g Database [7]. All these have, however, not gained broader adoption beyond the prototype level.

Colombo and Ferrari [8] prototypically implemented a comparable model for a document-based NoSQL database (MongoDB). Following the idea of query modification known from Hippocratic Databases, the enforcement of purpose-based access restrictions is here implemented as an additional, proxy-like component (called Mem — MongoDB enforcement monitor) that intercepts query messages and modifies them in line with respective policies. Due to this minimally invasive approach, any existing MongoDB installation can be enhanced with PBAC capabilities retroactively.

All the implementations mentioned above are, however, isolated solutions for exactly one particular database management system. Therefore, choosing one of the solutions for implementing PBAC in a real application necessarily imposes the risk of a vendor lock-in for the respective database system and, in particular, prevents a later switch to another one to better meet changed functional requirements, for example. Alternative approaches for implementing access control independently from the employed database through query rewrites of standard SQL (such as, for instance, QAPLA [20]), in turn, typically implement traditional, role-based access control models and do not pay regard to purpose-specific requirements.

2.2 Information Systems Architectures & Application Engineering

When mechanisms for privacy by design are to be implemented for actual use in real-world systems, sufficient consideration must also be given to the actualities and practices of systems and application engineering. Insofar, the well-established principle of layered architectures, commonly-used concepts regarding the interaction between application logic and databases, and actual practices for implementing access control in modern system architectures are of particular relevance here.

Layered Architectures. For reasons of scoping, to ensure maintainability and testability, and to allow for easy substitution of single components, modern application architectures are split into several layers, covering different functionalities and cleanly separarting them from each other. For instance, Alonso [2] distinguishes three core layers of information systems: The lowermost layer comprises all data sources – databases, file systems, data services, etc. – used by the information system and the management of these ressources. This layer is referred to as the "data" or "ressource management" layer. On top of this is the "application logic" layer, where all core functionalities of the information system are implemented. Finally, a "presentation" layer provides the functionalities necessary for users or other systems to communicate with the information system, including, for instance, the assembly of HTML-based representations of outputs from the application logic layer [2]. For the sake of clarity of argumentation, we refer to these layers as "data layer", "application layer", and "presentation layer" herein. Users, in turn, are not part of the information system itself but interact with it through interfaces provided by the presentation layer.

Given the multitude of functionalities to be implemented in the application layer, this can be split up into multiple sub-layers. Richards [27], for example, also recognizes a presentation layer on top and a data layer on the bottom of a layered architecture. Between these, however, he distinguishes between a "business layer" comprising the core functionalities and a "persistence layer" for mapping between application objects and database representations, for abstracting away the particularities of different databases from developers, etc.[3] This distinction is of particular relevance for the point to be made herein. For our purposes, we will thus in the following consider a "presentation", an "application", and a "data" layer with the application layer being split into a "business" and a "persistence" sublayer (see figure 1).

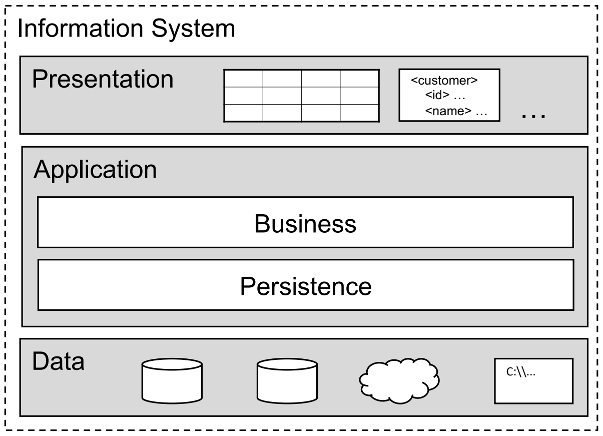

Figure 1: Layered architecture following [2, 27]

Database Abstraction through ORMs. Following the above layer structure, the persistence layer holds the functionality for mapping the business layer's object structure to a database representation, for querying the database and transforming results into objects usable in the business layer, and, importantly, for handling the communication with the database itself.

Especially for the mapping between object representations and relational databases, this functionality is in practice typically implemented using so-called "Object-Relational-Mappers" (ORMs). Following Fowler's ActiveRecord-pattern [11], such ORMs allow to define domain objects and the relations between them through comparably simple declarations. Developers only have to specify, for instance, that an object type "customer" holds a first- and a lastname, an address, etc., and that there is a 1-to-many relation between a customer and another object type "order". The ORM can then automatically create respective tables, fields, and relations in the database and allows developers to directly interact with abstract objects instead of having to execute low-level query statements.

[3]Finer layer granularities, for example separating different sub-domains vertically, as well as slightly differing layer conceptions [e.g., 12, 26] can also occur in practice.

Also, ORMs typically abstract away the particularities of different databases, facilitating the easy change between different ones.[4]

Much of the complex functionality of the persistence layer, including the ultimate communication with the database, is today thus often provided by ORMs. Besides implementation efficiency, this also avoids many of the pitfalls and errors typically arising from direct interactions with the database. ORM-based data access is therefore in many cases preferred over low-level database interactions.

Implementing Access Control in Practice. A core question for practically usable PBAC is where access control and the respective management of users and access rights are typically implemented in practice. Here, the distinction between application- and data layer is of particular relevance: Basically, most databases provide all functionalities (user management, access rights specificationn, etc.) necessary for implementing access control. These database-internal mechanisms are typically extended by existing PBAC approaches (see section 2.1 above). From the perspective of an application developer, making use of such extensions would thus require to rely upon the access control mechanisms of the database.

This would, however, be in stark contrast to the actual practice in application engineering. In reality, the management of users, roles, and access rights as well as their enforcement is usually *not* implemented on the data- but rather within the application layer. Typically, user objects, roles, access rights, etc. are created within the business sublayer and stored to the database using functionality from the persistence sublayer like any other object type.[5] In typical applications, the whole logic of access control thus only exists within the application layer. Respective functionalities from the data layer, in turn, are in most current architectures only used to authenticate *the application as a whole.* As Roichman and Gudes [28] put it, "[f]rom the database point of view, the user accessing the database is the super-user of the web application".[6]

This necessarily conflicts with the appraoch of data layer implementation followed by all existing PBAC proposals: When the whole application appears as just one user from the perspective of the data layer, implementing sophisticated PBAC functionality on this layer would hardly make sense. Using data layer access control within an application, in turn, would require to break establised and well-founded layering concepts, which is rarely practical. It is thus not surprising that existing PBAC proposals have so far not been broadly adopted in practice.

3 PRACTICE-ORIENTED IMPLEMENTATION OF PBAC

Taking the above-mentioned givens as reference points, we are striving for a practically valuable implementation of PBAC that provides necessary functionality and at the same time aligns well with established concepts and approaches from the actual practice of application engineering. For this aim, we identify crucial requirements first, followed by an outline of the most distinguishing concepts underlying our approach and a depiction and preliminary assessment of two validating implementations.

3.1 Requirements

Relevant requirements specific for a practically valuable PBAC mechanism arise for the foundational functionalities to be provided as well as for rather non-functional properties, mostly regarding the integration into the broader context of application engineering.

3.1.1 Functional Requirements.

Req. 1: Attribute-level Access Control. An ideal solution for purpose-based access control has to ensure that it is possible to specify different and numerous processing purposes for every detail of personal information. Furthermore, it must be possible to specify different purpose-related policies for different attributes so that, for instance, a customer's e-mail address may be used for the purpose of sending special offers while the telephone number may not. Therefore, the regulation of access to personal details needs to be organized at attribute-level. For access control solutions on the data layer, this model is called cell-level access control.

Req. 2: Hierarchies of Purposes. To be able to represent processing purposes in different levels of specificity, purposes should be organized in a hierarchical structure (e.g., a tree). Following this approach, it is possible to implement generalization and spezialization of purposes and categories thereof. Even though the GDPR requires purposes to be *specific*, it is observable in practice that institutions largely follow the approach of obtaining overly *broad consent.* Generalized categories and different levels of specificity may be a reasonable compromise here, allowing data controllers to consciously choose the level of detail depending on the particular use-case. At least, this proposes a viable alternative to the common usage of overly vague and broad purposes used today.

Req. 3: Flexibility of Purpose Vocabulary. In order to allow developers to use the proposed solution in a wide range of systems and applications, the specification of available purposes (the "purpose vocabulary") must be as open and flexible as possible and, in particular, not limited to a specific domain. For instance, relevant purposes will significantly differ between a shopping- and a health-related use-case. At the same time, striving for a universal, all-encompassing ontology of possible purposes that allows to fulfill any domain-specific (regulatory) requirement is illusory. Instead, it should thus be possible for developers to specify the vocabulary of available purposes individually, according to their particular domain- and application-specific needs. This should be possible independently from the solution's implementation itself through appropriate configuration mechanisms.

Req. 4: Distiction between Allowed & Prohibited Purposes. As mentioned earlier, it can be of great advantage to distinguish between permitted and prohibited processing purposes in a purpose-based access control model. In combination with the above-sketched possibility to represent hierarchies and relations between purposes, this also makes it possible to specify a generalized category of

[4]See, for instance, Souza [30], advocating the use of ORMs for reasons of easier implementation and automatic database schema generation. For the fact of ORMs being common practice especially applied in web application engineering, see also [16]
[5]Alternatively dedicated solutions for user and access management are also sometimes employed. These are, however, also embedded on the application layer.
[6]See also [19]: "the application [...] connects to the database as a single, highly privileged user", also referring to Oracle calling this approach the "One Big Application User" model. For the latter, see https://docs.oracle.com/cd/B28359_01/network.111/b28531/app_devs.htm#DBSEG98112

allowed purposes and exclude a single specific purpose from the same category. Besides allowing more compact and understandable access rules [7], this also eases the future handling of previously unforeseen, highly specific processing purposes.

3.1.2 Non-Functional Requirements.

Req. 5: Database Independence. To avoid application developers being worried about vendor lock-in to a particular database that provides PBAC functionality and therefore abstaining from adoption, the solution must provide database independence and be usable in conjunction with different databases. Also, it must not require a modified SQL dialect to be used in the application code (as it is, for instance, the case for [8, 17, 18]) to avoid incompatibilities with commonly used abstraction and optimization layers (e.g., ORMs). Instead, the solution should be "database agnostic" and require as few changes in established development stacks as possible.

Req. 6: Reasonable Performance Overhead. The GDPR obligates the use of technical measures for addresssing privacy principles "depending on the cost of implementation" in Art. 25. These cost of implementation must also include the additional operational cost arising from potential performance drops and, consequently, higher ressource consumption implied by a technical measure. The introduced performance overhead is thus of crucial importance for assessing whether a technical measure is to be applied or not: if it is reasonable (and the measure is actually effective in implementing the privacy principle), data controllers cannot justify to not use the measure. If, however, the overhead is excessive, data controllers are not obligated to use the measure, consequently leading to non-adoption of a technology, irrespectively of its effectiveness. Achieving a sufficiently small performance overhead is thus of utmost importance for our approach to gain practical relevance.

Req. 7: Developer-Friendliness. For similar reasons, a novel mechanism must also be developer-friendly and minimize implementation effort. Besides avoiding that data controllers might justify the non-use of PBAC with the effort-effect weighing from Art. 25 of the GDPR, developer-friendliness may also stipulate adoption in the positive sense: If integrating PBAC functionality into an application is easily done, developers might also use it out of intrinsic motivation or mere curiosity. Here, developer-friendliness not only refers to the amount or complexity of code to be programmed but also to factors like the coherent integration into established architectural models and development stacks (see section 2.2).

Req. 8: Reusability. In systems and application engineering as well as in software development in general, the principle of *Don't Repeat Yourself (DRY)* [14] is widely known and respected. Programmers reuse code since the early days of software development [29] and it was later-on established in academia and systems engineering research as an approach to faster development as well as better collaboration and education [4]. Thus, to support a wider adoption and foster a simplified integration of PBAC into applications and systems, the proposed solution should be implemented as a reusable artifact.

3.2 Core Concepts

While some the above-mentioned requirements can only be addressed on the concrete implementation level, others call for rather conceptual considerations – especially in the light of the givens laid out in section 2.2. We therefore propose a technical approach to PBAC that builds upon the following core concepts:

Application-Layer Implementation. Instead of implementing purpose-based access control on the data layer, we follow the approach of lifting respective functionality to the application layer. This significantly eases the use of our approach in practical implementation, allows for considerably better integration with established implementation practices and patterns, and, last but not least, provides flexibility regarding the database to be used. In particular, an application layer approach is in line with the established practice of implementing access control mechanisms mentioned above and allows developers to retain the lower-layer abstractions they are used to. The concept of application-layer implementation thus serves the above-mentioned requirements for developer-friendliness (**Req. 7**) and for database independence (**Req. 5**).

As a downside, our approach introduces the possibility of data being accessed directly from the database, circumventing the PBAC mechanism. We do, however, assess this limitation to be an acceptable price for achieving practical applicability of PBAC.

Database Independence through ORM-Extensions. As laid out above, ORMs are widely used in practice for abstracting away lower-layer interactions with databases and, in particular, to achieve database independence within the application layer. Developers of business-layer functionality typically use an ORM's unified programming interface for storing and accessing data. This makes the ORM a perfect spot for integrating PBAC functionality in a database-independent (**Req. 5**) and developer-friendly (**Req. 7**) way. Also, the functionalities required for PBAC from the perspective of business functionality (restrict access to data in line with specified purposes) as well as for the interaction with the data layer (purpose-based data filtering, automated generation of database structures) are quite similar across different application domains, making it particularly automation-friendly. We therefore propose to implement application-layer PBAC functionality through extensions of widely used ORMs.

As opposed to other possible (business-layer) implementations, integrating PBAC-related functionality into the ORM also provides high re-usability (**Req. 8**). An ORM extension introducing PBAC functionalities only has to be implemented once, building upon database abstractions already existing within the ORM, and can then be provided publicly for developers to use it within their own business-layer developments. Depending on the employed programming stack and ORM, it might even be implemented in a way that seamlessly "hooks" into an ORM so that it can also be used in future versions without necessarily requiring to be natively included, increasing re-usability even further.

A possible downside of the ORM-based approach could emerge as soon as a deeper integration between purpose-based and user-/role-based access rights becomes necessary (e.g., when only users holding a certain role should be able to access data for a particular purpose). As this downside does, however, also exist for data-layer

PBAC solutions and as the later addition of user-/role-based capabilities seems realistic, we see the above-mentioned advantages of an ORM-based implementation to clearly outweigh the downsides.

Specification of available purposes through configuration files. As laid out above, it must be possible to specify the available vocabulary of purposes flexibly, based on the specific givens of the particular use-case (**Req. 3**) and in a hierarchically structured manner (**Req. 2**). Of the various options for implementing the respective configuration functionality, we chose a developer-friendly approach of YAML-based configuration files. These allow to specify purposes in the required manner and are easily read- and conceivable for humans, facilitating constructive exchange with non-technical stakeholders (e.g., legal departments). Furthermore, developers are familiar with the YAML-format and can use mature tools (programming libraries, editors, ...) to process it (**Req. 7**). We foresee the ORM-integrated PBAC mechanism to perform all additionally required processing steps[7] automatically so that as soon as a configuration file is provided, developers can use respective purposes instantaneously.

Other, more complex approaches for providing purpose dictionaries can definitely be thought of, including rather formalized domain-specific languages [32] as well as more connected, service-based concepts. In particular, such approaches could allow for more detailed purpose specifications and ease the use of one purpose vocabulary in multiple applications within an organization or even industry. For our purpose of introducing and validating the general idea of application-layer PBAC, however, the file-based approach is more than sufficient and already goes beyond existing proposals in matters of developer-friendliness (**Req. 7**) and practical viability. Nonetheless, the mentioned approaches may be interesting candidates for later extensions.

3.3 Implementation 1: Ruby on Rails

As mentioned above, we implemented our approach for two different software frameworks widely used in real-world information systems engineering to demonstrate practical viability in matters of added programming effort as well as regarding the induced performance overhead. Without going into all details[8], both validating implementations shall be briefly depicted.

The first implementation is based on Ruby on Rails, an open source web application framework, which is – due to its maturity, the rich ecosystem of available extensions, and its particular fit for rapid prototyping – widely used by startups as well as by large online businesses. For interlinking application logic and database storage, applications commonly employ the *ActiveRecord* ORM provided in Ruby on Rails by default. ActiveRecord natively supports MySQL, PostgreSQL, and SQLite while other relational databases can be added through adapters.

Following our approach outlined above, we extended ActiveRecord with the automation and abstraction capabilities necessary to support PBAC with minimum programming efforts (**Req. 7**). In particular, we extended the Query API so that it supports purpose-enhanced queries out of the box. For instance, developers can, instead of a query statement like `User.where(firstname: "John")` now easily access user data for a particular purpose with `User.where (firstname: "John").for(<purpose ID>)` and the extended ORM will only return those entries for which access is to be granted for the particular purpose. Query statements without a given purpose are handled as if a non-existing purpose were passed, thus forcing developers to actually make purpose-aware queries.[9]

In addition, we also extended the tool included in Ruby on Rails for generating new default apps so that an example tree of available purposes (**Reqs. 2 & 3**) is included in the initial file structure. Finally, the so-called scaffolding mechanism, which generates default functionalities and methods for interacting with objects based on a simple model specification, was also extended so that it automatically creates purpose-aware method skeletons, purpose-related partner columns in the database for every attribute (storing the information for which purposes a particular field may be accessed, **Req. 1**), etc.[10] In the end, developers can simply initialize a new, purpose-aware web application with the same commands and workflows they are used to (**Req. 7**). The purpose-aware use of data, in turn, only requires minimal implementation overhead through the ORM's newly introduced `.for()`-notation and the whole variety of relational databases usable in Ruby on Rails is supported out of the box (**Req. 5**).

3.4 Implementation 2: Node.js

Node.js is another development framework widely used in the field of web applications. It uses Javascript and primarily focuses on the the API part and other server-side components of web applications while front-end aspects are typically covered by other, separate frameworks. Node.js also exhibits a rich ecosystem of extensions and tools, provides high performance and scalability, and integrates well into microservice environments, making it popular in startup as well as enterprise contexts. Among the different ORMs available in Node.js, *Sequelize* is the most mature and most widely used one. It also supports a wide variety of SQL databases including MySQL, PostgreSQL, MariaDB, etc., serving our goal of database independence (**Req. 5**).

Like Ruby on Rails' ActiveRecord, we extended Sequelize with the necessary capabilities for easily implementable PBAC.[11] In particular, Sequelize was enhanced to support automated generation of dedicated tables holding allowed purposes on a per-attribute- and a per-item basis (**Req. 1**)[12] based on simple model specifications as well as through newly introduced per-object-methods (e.g., `<object>.addPurpose('<purpose ID>')`). Available purposes are

[7]Depending on the implementation, this could include the generation of additional database tables or entries, for example

[8]including, for instance, the extension of purpose control to the initial storage of data, logging functionality to fulfill legal accountability obligations, automated management of retention periods, handling of "compatible purposes", etc.

[9]For more details, including the usage of hierarchical, two-dimensional purpose-specifications, see the code and documentation available online at https://github.com/PurposeOnRails/rails.

[10]Specifying prohibited purposes (Req. 4) is not yet supported at the moment. Even though mainly being a means of optimization and thus not hindering practical adoption, this might be a possible extension for future versions.

[11]The extension and a detailed documentation is publicly available as npm-package at https://www.npmjs.com/package/purposize .

[12]Again, specifying prohibited purposes (Req. 4) is not yet supported at the moment but might be added in future versions.

defined in a hierarchically structured set provided through a configuration file with compatibility relations[13] (**Reqs. 2 & 3**). Based on this, and comparably to the `.for()`-construct introduced to ActiveRecord above, our extension allows to introduce purposes to queries through an additional parameter, thus raising minimal implementation effort (**Req. 7**): Instead of `Customer.findAll()`, for instance, developers can simply use `Customer.findAll({purpose: '<purpose ID>'})` and the ORM returns all matching customer data. Due to the component architecture of Sequelize, most of these capabilities could easily be implemented by wrapping handlers around two core functions of Sequelize – `findAll` and `save` – and augmenting them with purpose-specific functionality.

3.5 Preliminary Performance Assessments

To avoid being on a fundamentally flawed path, we conducted preliminary performance assessments for our proof-of-concept implementations in line with established practices of security-related performance benchmarking [24, 25]. Noteworthily, these assessments were executed without any optimizations present in our implementations. Given the rather naïve internal data models currently employed and the multitude of further optimization possibilities, numbers shall thus explicitly be understood as rough indicators for the general practicability of our approach. Realistic impacts to be expected in practice after performance optimizations will be significantly lower than those given below.

For both stacks, we used a similar setup consisting of an AWS t2.large instance for the system under test. ActiveRecord and Sequelize were used to access an up-to-date PostgreSQL database with and without our PBAC extensions, respectively. For each implementation, we created the same dataset reflecting a typical fitness application scenario. This dataset comprised simple user data (1000 users, each containing name, username, date of birth, email address, gender, and phone number), daily step records (100 records per user, each record containing one value), and heart rate logs (100 records per user with 100 datapoints each). Both implementations were also configured with the same purpose-hierarchy of depth 3 containing an overall of 17 purposes attached to the different datasets.

Using built-in tools of the respective frameworks (Ruby on Rails and Node.js), these data were provided in two different ways: a data access API delivering plain JSON data directly from the ORM and a simple HTML representation of the same data rendered through the template engine, thus more closely resembling the complete stack of a real-world web application.

We benchmarked the achievable number of successful requests/s from an AWS t2.xlarge instance through ApacheBench[14] with a maximum of 25 concurrent requests, sending an overall of 60k realistic read requests per setting: 25k requests for profile data, 25k requests for 100 stepcounts (1 datapoint each) and 10k requests for 24 heart rate logs (100 datapoints each). To avoid benchmarking the client or other components of the setting than the actual effect of our ORM extensions, relevant parameters like network loads, client CPU load, etc. were carefully observed and found to stay within reasonable boundaries. Misleading results emanating from reduced result sizes were prevented by only accessing data for purposes that are actually allowed.

Ruby on Rails. For the Ruby on Rails implementation, we observed a performance drop of 24.5% when accessing user data and of 27.8% when retrieving step count data through the API interface while heart rate data exhibited significantly lower reductions of 10.4% compared to the native ORM. Given the rather static overhead for parsing and resolving purpose restrictions introduced by our PBAC extension per request independently from the result size and the non-existence of performance optimizations, these numbers depict an expectable system behavior. When we include the assembly of rather simple HTML pages in our benchmarks, in turn, the relative impact on overall system performance raised by our PBAC extension decreases to 17.8% for user data and 21.8% for step count data, while for heart rate data, the impact was below reasonable measuring accuracy (see table 1).

Even without any optimizations, these numbers already demonstrate the general viability of our approach and, at least for mid-sized and larger datasets, support its suitability in matters of performance overhead (**Req. 6**). The performance behavior for smaller datasets with multiple attributes, however, calls for further examination and optimizations – at least for use-cases where such queries are particularly relevant and executed without additional, more heavyweight processing steps such as assembling HTML representations. As vividly demonstrated by the numbers including the construction of (fairly simple) HTML documents, however, the impact of our PBAC extension can be considered significantly lower or even close to negligible for the overall performance of an entire application or information system.

Table 1: Benchmarking results (reqs/s) for Ruby on Rails and Node.js with and without our proof-of-concept PBAC extensions, differentiating between API access and inclusion of HTML rendering

	User Data	Steps	Heart Rate
API access			
Plain Ruby on Rails	419.1	123.4	78.4
Ruby on Rails with PBAC	316.5	89.2	70.2
Decrease	*24.5%*	*27.8%*	*10.4%*
Plain Node.js	1334.1	1034.8	321.1
Node.js with PBAC	913.7	655.3	257.3
Decrease	*31.5%*	*36.7%*	*19.9%*
Incl. HTML rendering			
Plain Ruby on Rails	235.2	58.6	3.7
Ruby on Rails with PBAC	193.3	45.8	3.7
Decrease	*17.8%*	*21.8%*	*<1.0%*
Plain Node.js	1176.7	574.9	61.6
Node.js with PBAC	845.5	445.3	56.4
Decrease	*28.2%*	*22.5%*	*8.5%*

[13]These compatibility relations might, besides constructing hierarchical dependencies, also be employed to represent the GDPR's concept of "compatible purposes".
[14]See http://httpd.apache.org/docs/2.4/programs/ab.html

Node.js. The relative numbers for the Node.js stack differ slightly from those for Ruby on Rails. Observed throughputs dropped by 31.5%, 36.7%, and 19.9% for API access and by 28.2%, 22.5%, and 8.5% with HTML construction included for user, step count, and heart rate data, respectively (see table 1). Even though confirming the strong dependency on load types and again highlighting the impact of additional processing steps such as HTML assembly, both effects are less dominant in our Node.js implementation. In addition, the relative impact of our PBAC extension is higher than the one observed for Ruby on Rails in all cases.

These findings can be explained by at least two facts: First, the well-known, significantly higher base performance of Node.js over Ruby on Rails expectedly increases the relative impact of the rather static per-request overhead for resolving access restrictions. Second, and presumably more decisively, significant parts of the overheads observed for Node.js can be ascribed to implementation-specific particularities. As mentioned above, our implementation internally uses several companion tables holding allowed purposes on a per-attribute- and per-item basis. This causes multiple additional (and performance-wise costly) `join`-operations across database tables for every single request. This foundational modeling choice is thus a clear candidate for future optimizations. Keeping this in mind, our Node.js implementation does, by and large, at least confirm the general viability of our approach across different development stacks and the possibility of providing capabilities for application-layer PBAC at minimum implementation overhead (**Req. 7**). Performance overheads (**Req. 6**), in turn, may be considered reasonable for certain use-cases (especially in the light of the significantly higher absolute performance of Node.js) but clearly deserve further examination and optimization.

4 DISCUSSION & FUTURE WORK

Of course, our aproach presented herein comes with drawbacks: First and foremost, anchoring purpose-based access control on the application- instead of on the data layer does not prevent data from being accessed directly from the database by, for instance, a malicious insider such as an application developer. Insofar, our approach necessarily leaves open a bigger "attack surface" for purpose-contradicting data access than other approaches like hippocratic databases. However, common application architectures employ a "single superuser" model for database access, while more fine-grained access control is only implemented on the application layer anyway (see section 2.2). In addition, the GDPR explicitly follows a risk- and, with regard to technical and organizational measures, effort-based approach. Even after years of research and development, proposed data-layer mechanisms for PBAC do not sufficiently integrate with prevailing practices and tools of application engineering and therefore remain overly burdensome to implement in practice. Any initiative to foster their actual use is therefore thwarted by rational effort-effect-considerations. Our approach of application-layer PBAC, in contrast, integrates well with these practices and tools and was shown to raise acceptable overheads in matters of programming effort and performance loss. Following the wisdom that "half a loaf is better than no bread", it therefore opens up the possibility to actually bring purpose-based access control into practice. Even with the remaining risk, this is clearly preferrable over having no practically usable PBAC at all.

Second, our focus has so far been on the rather general concept of lifting PBAC to the application layer for reasons of better alignment with practice and on the demonstration of its practical viability through two prototypical implementations. In so doing, several aspects necessarily had to be left uncovered for the moment. This particularly includes more detailed questions about purpose specifications, regarding integration with rather orthogonal role-based access control models (e.g., for allowing particular roles to access data for a certain purpose), and on possibilities for (more) conclusively integrating the GDPR's concept of "compatible purposes". In these regards, one next step will be to incorporate advanced concepts for hierarchical purpose specification such as YaPPL [32] into our approach. This will particularly serve the goal of being able to specify purposes in relation to different accessing parties, thus aligning better with legal requirements and increasing practical applicability even further. Also, our implementations currently lack the capabilities for combining allowed and prohibited purposes. We plan to include respective functionality in future versions.

Finally, the performance overheads introduced by our implementations were only briefly examined herein to at least preclude being on a fundamentally flawed path. Given the non-optimized nature of our proof-of-concept implementations, these assessments do not provide a sufficient basis for determining the actual overhead to be expected in real-world settings. We see significant optimization potential in our current implementations, ranging from less naïve internal data models to a reduction of added database interactions.[15] We therefore plan to implement respective optimizations in the future, accompanied by more extensive benchmarks.

Such benchmarks on the overhead introduced by a technical privacy measure are, last but not least, also of crucial importance for deciding about controllers' legal obligation to actually implement it, given that Art. 25 of the GDPR explicitly refers to "the cost of implementation" as a decisive factor. Here, the provision of experimentally gathered evidence demonstrating the reasonability of actual overheads raised by optimized versions may put data controllers under pressure to justify their disuse of available technical mechanisms for purpose-based access control.

5 CONCLUSION

In this paper, we introduced a novel, application-layer approach for addressing the privacy principle of purpose limitation technically. Even though respective proposals for purpose-based access control have been discussed for years, they clearly lack adoption in practice so far. We ascribe this to a conceptual misfit between existing PBAC proposals on the one and established concepts and practices from real-world application engineering on the other hand.

Taking into account these concepts and practices, we proposed an ORM-based approach to PBAC that aligns well with the actual practice of systems and application engineering and at the same time provides database independence, developer friendliness, and re-usability. We demonstrated the practical viability of our approach by means of two proof-of-concept implementations for

[15] Recall, e.g., the performance loss presumably stemming from additional joins in the Node.js implementation.

two different, widely used development stacks. Even though calling for further optimizations, these implementations already exhibit bearable performance overheads in some realistic scenarios and the additionally required programming effort is rather limited. Where overheads are currently substantial, we already foresee several optimizations to enhance practical viability of our approach even further.

The above-discussed open issues and optimization potentials notwithstanding, our concept of ORM-integrated purpose-based access control is thus a major step towards closing the gap between the state of the art of purpose-based access control and the actual practice of application engineering. Following the GDPR's obligation to implement *all* privacy principles technically as soon as effective means are available and implementation efforts are reasonable, it may thus pave the way for the privacy principle of purpose limitation to be widely addressed "by Design" in the future.

REFERENCES

[1] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2002. Hippocratic Databases. In *Proceedings of the 28th International Conference on Very Large Data Bases*. VLDB Endowment, Hong Kong, China, 143–154. http://dl.acm.org/citation.cfm?id=1287369.1287383

[2] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. 2004. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, Berlin Heidelberg. https://www.springer.com/de/book/9783540440086

[3] Jasmin Azemović. 2012. Data Privacy in SQL Server Based on Hippocratic Database Principles. http://blogs.msdn.com/b/mvpawardprogram/archive/2012/07/30/data-privacy-in-sql-server-based-on-hippocratic-database-principles.aspx

[4] V. R. Basili. 1989. Software development: a paradigm for the future. In *[1989] Proceedings of the Thirteenth Annual International Computer Software Applications Conference*. IEEE, Orlando, Florida, 471–485. https://doi.org/10.1109/CMPSAC.1989.65127

[5] Elisa Bertino. 2005. Purpose Based Access Control for Privacy Protection in Database Systems. In *Database Systems for Advanced Applications (Lecture Notes in Computer Science)*, Lizhu Zhou, Beng Chin Ooi, and Xiaofeng Meng (Eds.). Springer Berlin Heidelberg, Beijing, China, 2–2.

[6] Ji-Won Byun, Elisa Bertino, and Ninghui Li. 2005. Purpose Based Access Control of Complex Data for Privacy Protection. In *Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies (SACMAT '05)*. ACM, New York, NY, USA, 102–110. https://doi.org/10.1145/1063979.1063998

[7] Ji-Won Byun and Ninghui Li. 2008. Purpose Based Access Control for Privacy Protection in Relational Database Systems. *The VLDB Journal* 17, 4 (2008), 603–619.

[8] P. Colombo and E. Ferrari. 2017. Enhancing MongoDB with Purpose-Based Access Control. *IEEE Transactions on Dependable and Secure Computing* 14, 6 (Nov. 2017), 591–604. https://doi.org/10.1109/TDSC.2015.2497680

[9] European Parliament & Council. 1995. Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data. http://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:31995L0046&from=en

[10] European Parliament & Council. 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the Protection of Natural Persons with Regard to the Processing of Personal Data and on the Free Movement of Such Data, and Repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union* L 119/1 (2016), 1–88. http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A32016R0679

[11] Martin Fowler. 2003. *Patterns of Enterprise Application Architecture* (01 ed.). Addison Wesley, Boston.

[12] Martin Fowler. 2015. PresentationDomainDataLayering. https://martinfowler.com/bliki/PresentationDomainDataLayering.html

[13] FTC. 1998. Privacy Online: A Report to Congress. https://www.ftc.gov/sites/default/files/documents/reports/privacy-online-report-congress/priv-23a.pdf

[14] Andrew Hunt, David Thomas, and Ward Cunningham. 1999. *The Pragmatic Programmer. From Journeyman to Master* (01 ed.). Addison Wesley, Reading, Mass.

[15] ISO. 2011. ISO/IEC 29100:2011 - Information Technology – Security Techniques – Privacy Framework.

[16] Markku Laine, Denis Shestakov, Evgenia Litvinova, and Petri Vuorimaa. 2011. Toward Unified Web Application Development. *IT Professional* 13, 5 (Sept. 2011), 30–36. https://doi.org/10.1109/MITP.2011.55

[17] Yasin Laura-Silva and Walid Aref. 2006. Realizing Privacy-Preserving Features in Hippocratic Databases. http://docs.lib.purdue.edu/cstech/1665

[18] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovac, Raghu Ramakrishnan, Yirong Xu, and David DeWitt. 2004. Limiting Disclosure in Hippocratic Databases. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases-Volume 30*. VLDB Endowment, Toronto, Canada, 108–119. http://dl.acm.org/citation.cfm?id=1316701

[19] Qiang Lin. 2004. Defense In-Depth to Achieve " Unbreakable " Database Security. In *Proceedings of the 2nd International Conference on Information Technology for Application*. unknown, Harbin, China, 386–390.

[20] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. 2017. Qapla: Policy compliance for database-backed systems. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1463–1479.

[21] OECD. 1980. OECD Guidelines on the Protection of Privacy and Transborder Flows of Personal Data.

[22] OECD. 2013. THE OECD PRIVACY FRAMEWORK. http://www.oecd.org/sti/ieconomy/oecd_privacy_framework.pdf

[23] J. Padma, Y. N. Silva, M. U. Arshad, and W. G. Aref. 2009. Hippocratic PostgreSQL. In *Procceedings of the 2009 IEEE 25th International Conference on Data Engineering*. IEEE, Shanghai, China, 1555–1558. https://doi.org/10.1109/ICDE.2009.126

[24] Frank Pallas, David Bermbach, Steffen Müller, and Stefan Tai. 2017. Evidence-Based Security Configurations for Cloud Datastores. In *Proceedings of the the 32nd ACM Symposium on Applied Computing*. ACM, Marrakech, Morocco, 424–430.

[25] Frank Pallas and Martin Grambow. 2018. Three Tales of Disillusion: Benchmarking Property Preserving Encryption Schemes. In *15th International Conference on Trust, Privacy and Security in Digital Business - TrustBus 2018*. Springer, Regensburg, Germany, 39–54.

[26] Pethuru Raj, Anupama Raman, and Harihara Subramanian. 2017. *Architectural Patterns: Uncover essential patterns in the most indispensable realm of enterprise architecture* (1 ed.). Packt Publishing, Birmingham, UK.

[27] Mark Richards. 2015. *Software Architecture Patterns*. O'Reilly Media, Inc., Sebastopol, USA.

[28] Alex Roichman and Ehud Gudes. 2007. Fine-grained Access Control to Web Databases. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT '07)*. ACM, Sophia Antipolis, France, 31–40. https://doi.org/10.1145/1266840.1266846

[29] Johannes Sametinger. 1997. Software Reuse. In *Software Engineering with Reusable Components*, Johannes Sametinger (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 9–19. https://doi.org/10.1007/978-3-662-03345-6_2

[30] Vítor Estêvão Silva Souza and Ricardo de Almeida Falbo. 2005. An Agile Approach for Web Systems Engineering. In *Proceedings of the 11th Brazilian Symposium on Multimedia and the Web (WebMedia '05)*. ACM, Pocos de Caldas - Minas Gerais, Brazil, 1–3. https://doi.org/10.1145/1114223.1114237

[31] Sarah Spiekermann and Lorrie Faith Cranor. 2008. Engineering Privacy. *IEEE Transactions on Software Engineering* 35, 1 (2008), 67–82.

[32] Max-R Ulbricht and Frank Pallas. 2018. YaPPL-A Lightweight Privacy Preference Language for Legally Sufficient and Automated Consent Provision in IoT Scenarios. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, Barcelona, Spain, 329–344.