Universidade de Aveiro
2021

Diogo José Domingues Regateiro

Acesso Remoto Dinâmico e Seguro a Bases de Dados com Integração de Políticas de Acesso Suave

Dynamic and Secure Remote Database Access with Soft Access Policies Integration

Universidade de Aveiro
2021

Diogo José Domingues Regateiro

Acesso Remoto Dinâmico e Seguro a Bases de Dados com Integração de Políticas de Acesso Suave

Dynamic and Secure Remote Database Access with Soft Access Policies Integration

Tese apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Doutor em Informática, realizada sob a orientação científica do Doutor Óscar Mortágua Pereira, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro e do Doutor Rui Aguiar, Professor associado com agregação do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Apoio financeiro da FCT no âmbito da bolsa SFRH/BD/109911/2015.

Dedico este trabalho a todos os que me ajudaram nestes últimos anos.

o júri

presidente

Doutor Victor Miguel Carneiro de Sousa Ferreira
Professor Catedrático, Universidade de Aveiro

vogais

Doutor Marco Paulo Amorim Vieira
Professor Catedrático, Universidade de Coimbra

Doutor Rui Luís Andrade Aguiar (Coorientador)
Professor Catedrático, Universidade de Aveiro

Doutor Jorge Miguel de Matos Sousa Pinto
Professor Associado com Agregação, Universidade do Minho

Doutor José Manuel Matos Moreira
Professor Auxiliar, Universidade de Aveiro

Doutor Paulo Jorge Machado Oliveira
Professor Adjunto, Instituto Superior de Engenharia do Porto

agradecimentos

Gostaria de agradecer ao meu orientador Prof. Dr. Óscar Mortágua Pereira e ao meu co-orientador Prof. Dr. Rui L. Aguiar, por me guiarem pelo caminho correto durante todos estes anos. Gostaria também de agradecer ao Instituto de Telecomunicações por me proporcionar um ambiente de trabalho estimulante e as ferramentas necessárias para o meu trabalho de investigação e desenvolvimento desta tese.

palavras-chave

segurança informática, controlo de acesso, controlo de acesso difuso, arquitectura de software, engenharia de software, sistemas distribuídos, sistemas difusos, middleware, bases de dados, engenharia de software baseada em componentes, engenharia de software automatizada, otimização de algoritmos.

resumo

A quantidade de dados criados e partilhados tem crescido nos últimos anos, em parte graças às redes sociais e à proliferação dos dispositivos inteligentes. A gestão do armazenamento e processamento destes dados pode fornecer uma vantagem competitiva quando usados para criar novos serviços, para melhorar a publicidade direcionada, etc. Para atingir este objetivo, os dados devem ser acedidos e processados. Quando as aplicações que acedem a estes dados são desenvolvidos, ferramentas como Java Database Connectivity, ADO.NET e Hibernate são normalmente utilizados. No entanto, embora estas ferramentas tenham como objetivo preencher a lacuna entre as bases de dados e o paradigma da programação orientada por objetos, elas concentram-se apenas na questão da conectividade. Isto aumenta o tempo de desenvolvimento, pois os programadores precisam dominar as políticas de acesso para escrever consultas corretas. Além disso, quando usado em aplicações de bases de dados em ambientes não controlados, surgem outros problemas, como roubo de credenciais da base de dados; autenticação de aplicações; autorização e auditoria de grandes grupos de novos utilizadores que procuram acesso aos dados, potencialmente com requisitos vagos; escuta da rede para obtenção de dados e credenciais; personificação de servidores de bases de dados para modificação de dados; manipulação de aplicações para acesso ilimitado à base de dados e divulgação de dados; etc.

Uma arquitetura capaz de resolver esses problemas é necessária para construir um conjunto confiável de soluções de controlo de acesso, para expandir e simplificar os cenários de aplicação destes sistemas. O objetivo, então, é proteger o acesso remoto a bases de dados, uma vez que as aplicações de bases de dados podem ser usados em ambientes de difícil controlo e o acesso físico às máquinas/rede nem sempre está protegido. Adicionalmente, o processo de autorização deve conceder dinamicamente as permissões adequadas aos utilizadores que não foram explicitamente autorizados para suportar grupos grandes de utilizadores que procuram aceder aos dados. Isto inclui cenários em que a definição dos requisitos de acesso é difícil devido à sua imprecisão, geralmente exigindo um especialista em segurança para autorizar cada utilizador individualmente. Este objetivo é atingido no processo de decisão de controlo de acesso com a integração e auditaria das políticas de acesso suaves baseadas na teoria de conjuntos difusos. Uma prova de conceito desta arquitetura é fornecida em conjunto com uma avaliação funcional e de desempenho.

keywords

information security, access control, fuzzy access control, software architecture, software engineering, distributed systems, fuzzy systems, middleware, databases, component-based software engineering, automated-software-engineering, algorithm optimization.

abstract

The amount of data being created and shared has grown greatly in recent years, thanks in part to social media and the growth of smart devices. Managing the storage and processing of this data can give a competitive edge when used to create new services, to enhance targeted advertising, etc. To achieve this, the data must be accessed and processed. When applications that access this data are developed, tools such as Java Database Connectivity, ADO.NET and Hibernate are typically used. However, while these tools aim to bridge the gap between databases and the object-oriented programming paradigm, they focus only on the connectivity issue. This leads to increased development time as developers need to master the access policies to write correct queries. Moreover, when used in database applications within non-controlled environments, other issues emerge such as database credentials theft; application authentication; authorization and auditing of large groups of new users seeking access to data, potentially with vague requirements; network eavesdropping for data and credential disclosure; impersonating database servers for data modification; application tampering for unrestricted database access and data disclosure; etc.

Therefore, an architecture capable of addressing these issues is necessary to build a reliable set of access control solutions to expand and simplify the application scenarios of access control systems. The objective, then, is to secure the remote access to databases, since database applications may be used in hard-to-control environments and physical access to the host machines/network may not be always protected. Furthermore, the authorization process should dynamically grant the appropriate permissions to users that have not been explicitly authorized to handle large groups seeking access to data. This includes scenarios where the definition of the access requirements is difficult due to their vagueness, usually requiring a security expert to authorize each user individually. This is achieved by integrating and auditing soft access policies based on fuzzy set theory in the access control decision-making process. A proof-of-concept of this architecture is provided alongside a functional and performance assessment.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

ABAC	**A**ttribute **B**ased **A**ccess **C**ontrol
API	**A**pplication **P**rogramming **I**nterface
BDFIS	**B**inary **D**ecision **FIS**
COG	**C**enter **O**f **G**ravity
COGS	**C**enter **O**f **G**ravity for **S**ingletons
CRUD	**C**reate, **R**ead, **U**pdate, **D**elete
FCL	**F**uzzy **C**ontrol **L**anguage
FDC	**F**uzzy **D**ecision **C**omponent
FIS	**F**uzzy **I**nference **S**ystem
IDE	**I**ntegrated **D**evelopment **E**nvironment
IoT	**I**nternet **o**f **T**hings
JDBC	**J**ava **D**ata**B**ase **C**onnectivity
LINQ	**L**anguage **In**tegrated **Q**uery
NoSQL	**N**ot **o**nly **SQL**
ORM	**O**bject-**R**elational **Mapping**
PAP	**P**olicy **A**dministration **P**oint
PDP	**P**olicy **D**ecision **P**oint
PEP	**P**olicy **E**nforcement **P**oint
PIP	**P**olicy **I**nformation **P**oint
PRP	**P**olicy **R**etrieval **P**oint
RADIUS	**R**emote **A**uthentication **D**ial-**I**n **U**ser **S**ervice
RBAC	**R**ole **B**ased **A**ccess **C**ontrol
RMI	**R**emote **M**ethod **I**nvocation
S-DiSACA	**S**ecure, Dynamic and **Di**stributed **S**oft **A**ccess **C**ontrol **A**rchitecture
S-DRACA	**S**ecure, Dynamic and **D**istributed **R**ole-based **A**ccess **C**ontrol **A**rchitecture
SQL	**S**tructured **Q**uery **L**anguage
SSL	**S**ecure **S**ockets **L**ayer
TLS	**T**ransport **L**ayer **S**security
VPN	**V**irtual **P**rivate **N**etwork
XACML	e**X**tensible **A**ccess **C**ontrol **M**arkup **L**anguage
XML	e**X**tensible **M**arkup **L**anguage

Chapter 1

Introduction

Access control has always been an important feature on any system, be it physical or digital, as it restricts the access to a location or resource in a controlled and selective manner [1]. In the digital landscape, the storage and processing of data by businesses is a crucial step towards providing better services and gaining a competitive edge. Thus, being able to create new services and applications fast and effectively based on this data is an important aspect to master.

Thus, these services and applications must be able to access and operate on the data. This process is usually accomplished using predefined operations, such as create, read, update, and delete (CRUD) expressions, that are executed through database access tools such as Java Database Connectivity (JDBC) [2], Language Integrated Query (LINQ) [3], ADO.NET [4] and Hibernate [5], among others.

On the one hand, most of these tools allow developers to write and execute their operations, however, they forsake the access policies set in place. This may lead to authorization errors being issued by the database during execution if the operations access unauthorized data. Thus, developers must master the database schemas and what they are authorized to do with the data, leading to increased development and quality assurance time. On the other hand, tools that feature object-relational mapping, such as Hibernate, map the underlying database schema and relations to objects in an object-oriented domain model. These tools allow developers to write applications without having to master the database schema but at the cost of disclosing the schema in the application code. This information can be used to learn what information is being processed by a business or even to carry out attacks on the database. Furthermore, network eavesdropping, server impersonation and credential theft are issues that are commonly not addressed by these tools, and the issue with the access control policies not being reflected on the data access application programming interface (API) also remains.

To add to these issues, in recent years the quantity and complexity of data that needs to be stored and processed have increased considerably. This has led many new application scenarios, each with their specific data access policies and security requirements, to use more specialized tools to access and process data such as Not Only SQL (NoSQL) data stores [6]–[8] and MapReduce frameworks [9]–[11]. Moreover, more and more subjects are interested in accessing data, be it for commercial purposes, research, hobbyism, etc., which is also a challenging aspect to manage and handle effectively. Authorizing and auditing the access to an ever-increasing amount of information by subjects with

different backgrounds and purposes for the data is complicated, and often relies on soft requirements that are not easily translated into traditional access control policies.

Put together, a path leading to better development tools and security deployment can be found by tackling these issues.

1.1 Motivation

Since the aforementioned issues emerge during the development of database applications and the enforcement of access control policies, let us consider a database application development scenario. In this scenario, an application is being developed to directly access and manipulate the data that is stored in a database, which contains a schema that may evolve if the requirements change.

The application developer may typically either write the queries that access and manipulate the data manually or use an object-relational mapping (ORM) tool such as Hibernate to create a model of the database schema in the application code. The first option, while granting the developer greater control over the model, comes with many drawbacks. First, the developer is required to master the database schema to write the queries. Second, he must also be aware of the application permissions to not write queries that attempt to access unauthorized data. Third, a lot of development time is spent writing and debugging the code that connects and manipulates the data in the database. Finally, if the schema in the database ever changes, it may require the developer to modify the queries manually to comply with the new schema. The second option can reduce the development time by automatically creating the code that connects to the database, however, the developer must be careful to try not to access unauthorized data, as ORM tools do not ensure the enforcement of the database access permissions on the code layer.

The security of the solution must also be considered. Typically, database connectivity tools do not encrypt the connection to the database, which can be problematic if the applications that use them are running on machines located on (semi-)public locations. This fact can easily lead to database credentials being stolen via network eavesdropping to potentiate a later attack on the database itself. Furthermore, applications may have to use hard-coded queries to perform their data manipulation needs. A malicious user could tamper with these queries to achieve various goals in multiple ways, from the manipulation of the parameters passed to them to outright changing the entire query.

Other common scenarios include the intention to monetize the data owned, either because it holds scientific importance or because can be used for marketing purposes, or building a large set of community-managed data. This may lead to many different subjects requesting access to the data, each with slightly different purposes for it. The level of access to the data may change depending on the subject and may depend on policies that are not as clear cut as traditional access control policies usually allow. This puts a large amount of pressure on the security experts that normally have to grant access to each subject manually.

Thus, the central research focus is to create an architecture solution capable of generating secure data access APIs tailored to each subject that is also capable of enforcing access control policies with soft requirements to handle new subjects seeking access to the data. By taking the access control

policies into account when generating the data access APIs, a developer would no longer have to master the database schema or its access permissions. However, this architecture is not limited to generating these APIs. Additional security features can be built on over it, such as validating the queries and the parameters used to access the data, communication encryption with the database, etc. Furthermore, its access control decision-making system should be able to quickly map permissions to new subjects requesting access to data, instead of having permissions mapped to known users *a-priori*. This would, in turn, allow the data access APIs to be tailored to each subject at runtime without the constant need for a security expert to manually evaluate which permissions to grant, increasing the data availability and decreasing the management complexity.

1.2 Objectives

To actualize such an architecture, a subject should be able to extract a set of standard interfaces that it can use to access the data, which are only implemented at runtime. The extraction process validates the subject and obtains the permissions that were granted to it. Each permission is associated with a set of operations so that the data access interfaces can be generated containing only the operations allowed to that subject. Thus, the subject can utilize the interfaces in the application during development. When the subject runs the application, the interfaces are implemented and dynamically loaded, so that modifying the interfaces before execution is not possible.

Since the operations are predefined and the subject only has access to those that it is allowed to execute, this approach decreases the time spent in quality assurance to find and fix related errors. Moreover, the developers no longer have to master the database schema to write the operations. However, this architecture is intended to be deployed as an access control solution. While many standard security features can be trivially implemented (encrypted communication, subject authentication, etc.) researching solutions to enhance their effectiveness, usability, or adapting them to be deployed in scenarios with special security requirements is worthwhile. Such scenarios include community-managed data (e.g. Wikipedia), where data can be accessed and modified by anyone. Manual authorization of every subject requesting access to modify data is impractical given a large number of requests, however, having no access control system in place leads to data being modified maliciously (i.e. vandalism [12]). In this light, it is also important to determine how access control policies based on soft requirements can be defined to reach a balance: to allow subjects to be granted access to data in real-time without it being easily vandalized; and without needing constant manual authorization.

As such, several research goals were identified to realize such an architecture:

1. Automate the extraction of the interfaces to guarantee the latest access control policies are always reflected;

2. Use the access interfaces as parameters for operations to ensure that the values passed are correct;

3. Push the database operations to the server-side so that a subject cannot modify them on the application;
4. Push the database credentials to the server-side so that they cannot be stolen from the client applications;
5. Implement a secure communication channel based on pre-shared keys for situations where certification authorities cannot be trusted;
6. Allow defining the order in which the operations should be executed to follow predefined use cases;
7. Support for different databases and adjust the interface generation and implementation process for generic APIs;
8. Design an access control decision-making component that supports dynamic attribution of permissions and soft requirements.

Goal 1 is a core step to the architecture that aims to allow access control policies to be modified and reflected automatically on the client application development code. This way, developers are not required to keep updating the data access interfaces manually. Once an access control policy is modified, an integrated development environment can highlight the existing errors due to the change in permissions during compilation, instead of having the errors appear only at runtime. This not only reduces the burden on the application developers by not having to master the data access operations they are authorized to use but also reduces the time spent debugging the application and increases the overall quality of the final product.

Goal 2 aims to prevent applications from entering invalid or incorrect values as parameters into operations. Consider two operations identified by the labels A and B. Furthermore, operation B requires a parameter whose value can be obtained by executing operation A. Instead of relying on the application to get the value from operation A and passing it to operation B, and possibly changing it with malicious intent, the application selects a data entry from the operation A result set. Then, when the operation B execution is requested, the architecture can take the value selected in the result set and apply it as a parameter for operation B.

The intent of goal 3 is to prevent malicious subjects from modifying the operations that are executed on the data. Since the operations were initially sent to the client application, which then connected directly to the database through the generated data access interfaces, the data access logic was not secure. While the data access logic was only implemented at runtime, reflection mechanisms can be used to modify this it at runtime. Thus, by creating functions that execute the operations on the database and sending to the client application tokens that reference said functions (e.g. the function name), the data access logic is kept on the server. This change means that instead of using the actual operation, the client requests their execution using the token that cannot be modified without being rejected by the server.

Goal 4 is similar to goal 3, except the objective is to protect the database credentials. This related to an issue found primarily in applications used within a business, where they connect to a database

directly and store the credentials with them. However, some machines may operate in semi-public locations such as a reception desk, meaning that any user could potentially get access to it. Instead of having the client application connect to the database directly, which would require the database credentials to be known, the credentials should be stored on a server-side application that connects to the database in its stead. This way, if the client application is attacked and exploited, the database credentials remain safe.

Goal 5 comes as an attempt to enable secure communication between components without having to rely on third-parties. The most common method of securing communication is through the use of digital certificates, which allows a client and a server to establish a secure communication channel. For these certificates to be trustworthy, they must be signed by a certification authority to prevent a malicious user from creating its own certificate and stating it belongs to the server. However, a certification authority must be trusted to not issue a certificate that allows a malicious user to impersonate the server. This trust dependency may not be acceptable in every scenario. Furthermore, the issuing of these certificates may also not come cheap depending on the certification authority. Thus, researching another method of establishing secure communication channels is desired.

The purpose of goal 6 is to incorporate the use case logic in the access control layer. Since data access should not be done outside the established use cases, each use case could be defined as a sequence of operations. These sequences are then incorporated into generated interfaces and the developers are forced to follow them to access the data. This prevents operations from being executed in unexpected arrangements that could potentially disclose sensitive information.

Goal 7 is concerned with support custom data access APIs within the architecture. This approach is intended to enable support for any database, including non-relational ones which have acquired a larger presence in recent years. Since the interfaces being generated were initially based on JDBC and were therefore built to access relational databases, these interfaces and their definition process have to be overhauled to broaden the applicability of the automatic generation of data access interfaces to more scenarios.

Finally, goal 8 aims to evolve the architecture to be able to dynamically grant or deny permissions to subjects, possibly based on soft requirements, as they issue access requests. Without this, every time a new subject requests access to data it must be manually granted the permission to do so. This delays access to the data and increases management complexity. This complexity can be so great that some scenarios where the data is not sensitive just grant access to everyone and revokes the access later if an issue arises. An example of such a scenario is the Wikipedia, as it allows anyone to edit its pages, and later rollbacks changes and revokes access to individual contributors when malevolent modifications (vandalism) are made. This example also showcases a possible soft requirement, where users that are considered to be vandals should not be granted permission to modify pages.

The initial architecture and goals 1 to 6 were the focus of existing previous works [13]–[18], culminating in the Secure, Dynamic and Distributed Role-based Access Control Architecture (S-DRACA). In this thesis, the work on goals 2 to 6 is continued while goals 7 and 8 are first achieved. Thus, this thesis focuses on two major sections. The first section details the improvements achieved in the architecture itself, which includes how the operations and their parameters can be protected from outside tampering, how operations can be sequenced to prevent operations to be used together in

unexpected ways that may disclose sensitive data, database credential protection from being disclosed from the client applications and overall communication security. The second section focuses on the incorporation of fuzzy logic for access control decision making, which includes the development of a generic fuzzy inference system capable of making binary decisions that can be configured using definition files, and the development of an optimized auditing algorithm for this system to ensure the correctness of its policies.

1.3 Research Questions

With the motivation layed out, the goals that were set for the data access architecture naturally raise several questions. Some of these questions were addressed in the previous iteration of this work, but not all. Furthermore, the intent of supporting soft access control requirements adds to these questions as well to further expand the application scenarios of access control systems. After analysing every goal set forth, the following research questions remain to be answered:

- **RQ1**: How to disconnect the architecture from the data storage solution so that it is no longer tied to relational databases while providing developers with the same error-free data access interface?
- **RQ2**: Can the credentials to the data store be protected in such a way that internal attacks on the server-side cannot expose them?
- **RQ3**: How to design the custom data access interfaces so that developers can follow them easily while developing applications and integrating the sequences of operations with the operation parameter protection?
- **RQ4**: Since the underlying data stores may not have any access control features, how can the access control policies be designed so that they can be modified at runtime, new users that request access to data have an access decision made automatically, and soft data access requirements can be satisfied?
- **RQ5**: Can soft access control policies be audited for correctness before deployment?
- **RQ6**: How do these changes impact the security of the system and the confidentiality of the data?

With RQ1 the intent is to generalize the architecture to support any underlying data storage solution, whilst providing an access control error-free API to access the data contained within. This implies creating an abstraction layer that can access the specific data store used and obtain the required data, Furthermore, the client-side API interface and implementation mechanics need to be overhauled to allow data access through this API and not just mediated directly to a relational database.

RQ2 intends to find a way to protect the data store access credentials. Since the server has to host a data access API, the data store credentials must be present to enable this. However, this opens the door to potentially malicious users to obtain these credentials from the server in internal attacks.

RQ3 aims to research a way to make customized, easy to follow data access API interfaces available to the developers while preventing potential malicious users from modifying the data access logic. The previous work already had a basic solution to this, which consisted of sending a token that identified existing stored procedures in the relational database that could be executed. However, this solution is no longer valid given RQ1. Another aspect related to this research is how to create protected parameters that contain data previously obtained from the API. Since data is obtained under specific data access control policies, the data that is obtained can be used to access other related data. This enables the client applications to potentially modify the values for these parameters, which could allow them to access more data than intended. Parameters should be able to flow in the sequences without being modifiable.

RQ4 focuses on the access control policies, which were still statically defined in the previous work. Since the underlying data store is unknown, the architecture needs to deploy its specific access control model and allow policies to be modified at runtime. To remain relevant with recent advances in data storage, processing, and access requirements referenced in this chapter, the access control model must be able to loosely map users to data access permissions. Thus, when a new user requests access to some data, its parameters can be used to determine if that user has permission to do so or not, meaning that each user does not have to be given permission explicitly. Furthermore, the application of this architecture in scenarios that cannot deploy existing access control models due to soft access control requirements is also intended to be supported.

RQ5 targets a drawback of supporting application scenarios that have soft access control requirements that was discovered while researching RQ4. Since soft data access requirements cannot be crisply defined, there is always a degree of vagueness in the access control rules that are defined in the associated policies. Therefore, a mechanism or algorithm capable of analysing the policies written for this architecture that outputs the groups of input values that grant or deny permissions is important to increase the trustworthiness of the access control policies.

Finally, RQ6 aims to highlight other security and confidentiality issues that arise given the characteristics of the proposed enhancements to the goals identified for the architecture and to provide possible solutions.

1.4 Contributions

The search for the answers to the research questions led the work in this thesis through several lines of research: from the analysis of existing access control models [19]–[25]; the proposal of different methods of enforcing access control policies [18], [26]–[32]; and the study of several database and access control related security issues [25], [33]–[36].

These lines of research resulted in several contributions:

1. A formal definition of the sequence access control model that governs the order in which operations can be executed on the data store [31], which evolves the work presented in [26].

2. A method to protect data store access credentials so a malicious user has to break into two different servers to acquire them [35], [36].

3. Further enhancements to security features [34] and formalization of others [28].
4. The development of a Binary Decision Fuzzy Inference System (BDFIS) [32] that is able to handle soft access control requirements during the decision making process.
5. The design of a Secure, Dynamic and Distributed Soft Access Control Architecture (S-DiSACA) that evolves a previous work [18] and incorporates the BDFIS as its decision-making mechanism.
6. An optimized search algorithm to audit the correctness of the policies built for the BDFIS [37] and other non-standard decision-making methodologies [29], which is arguably the most important contribution made with this thesis.
7. Surveying and addressing security and data confidentiality issues by leveraging the work done with the S-DiSACA [30], [38] and other collaboration work [25], [39].

1.5 Dissertation Outline

This dissertation is divided as follows: section 2 provides the background on the previous work leveraged in this thesis and on fuzzy logic and fuzzy set theory; section 3 provides the necessary insight into the current state of the art regarding models that deal with vague concepts and/or deal with dynamically assigned permissions; section 4 details the security requirements and associated issues with the type of access control system presented, alongside the designed first-step solutions for them; section 5 introduces the BDFIS and explore the security issues that arise from using such a system; section 6 presents the proof-of-concept of the S-DiSACA along with some performance evaluation; finally, section 7 discusses the goals and results of the thesis.

Chapter 2

Background

In this chapter, the necessary background information to fully understand the remainder of this work is provided.

The background information is divided into two sections: the previous work in section 2.1, which is used as a base for many of the security-related contributions achieved in this work; and the fuzzy sets and fuzzy logic information in section 2.2, which introduces the most common approaches to incorporate soft requirements in access control models.

2.1 Previous Work

The previous work this thesis builds upon is an access control architecture called Secure, Dynamic and Distributed Role-based Access Control Architecture (S-DRACA) [18], which was developed targeting applications that use tools such as Java Database Connectivity [2], Hibernate [5] or ADO.NET [4] to access data stored in databases using Role-Based Access Control (RBAC). The main issue was that while these tools bridge the gap between the relational databases and the object-oriented programming paradigms, they do not incorporate into their interfaces the applied access control policies. Thus, to use these tools, developers must master the applied access control policies to be able to write correct applications.

Figure 2.1 shows the S-DRACA and its usage is divided into three major phases:

1. **Compilation Phase**, where data access interfaces called Business Schemas are generated from the applied access control policies.

2. **Pre-Execution Phase**, where the Business Schemas are implemented from the applied access control policies.

3. **Execution Phase**, where the implemented Business Schemas are used to access the data, respecting the applied access control policies automatically.

The Business Schemas mirror the application programming interface (API) provided by the standard tools used to access the data in relational databases, the difference being that the data access functions that were not allowed to be executed by the target application did not exist in the interface of

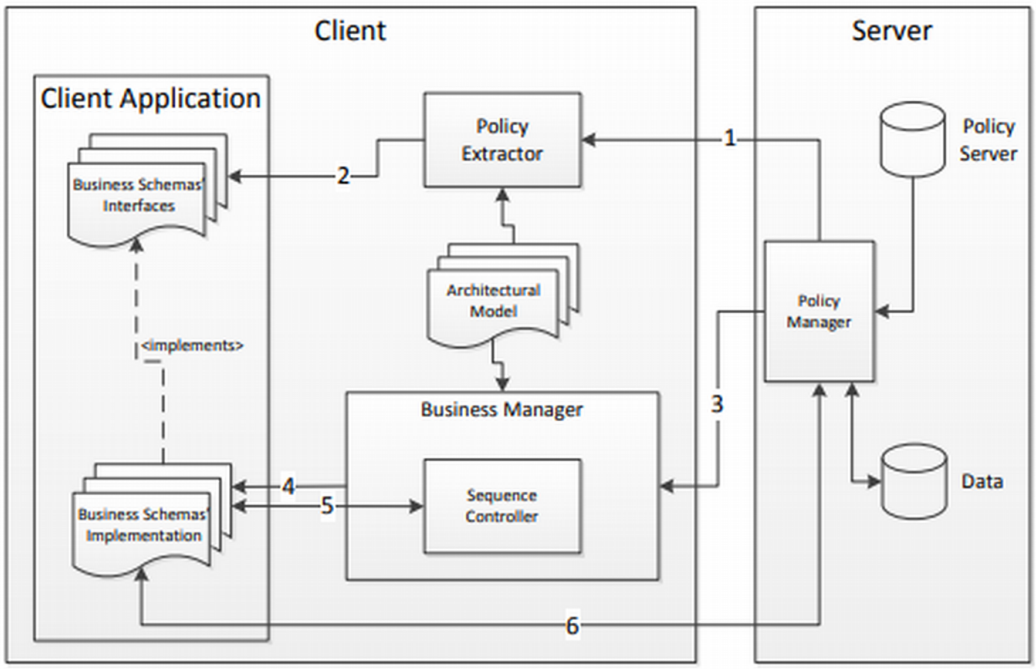


FIGURE 2.1: S-DRACA overview.

the Business Schema. Thus, the developers did not have to master the applied access control policies, lowering the time spent in development and debugging.

A short description of each phase is provided next, which explains the function of each block in Figure 2.1.

2.1.1 Compilation Phase

In the compilation phase, the developers configure the application with the credentials granted by the server. These credentials are used to authenticate the application and determine the associated role and to determine which Business Schemas to use.

Since different tables in the database can have several create, read, update, and delete (CRUD) expressions associated with them (to select different fields, to filter results, etc.), each CRUD expression is handled by a unique Business Schema. This Business Schema then allows the application to set parameters (if any) to queries, and depending on the CRUD expression to explore the results, update/insert/delete rows, etc.

Therefore, at compile-time, the compiler uses the *Policy Extractor* (see Figure 2.1) to first authenticate the application with the *Policy Manager* and then request the Business Schemas metadata that the application is allowed to use (1). Then, using the architectural model and the received metadata, it automatically generates the interfaces of the Business Schemas (2). The architectural model differs

between programming languages and defines the data access functions and how each one should be implemented.

The *Policy Manager* resides on the server, and in this phase, it accesses the *Policy Server* database where the access control policies are stored (including the Business Schema metadata, CRUD expressions and application information) to authenticate the application and send the data required by the *Policy Extractor* to generate the Business Schema interfaces.

The key difference from similar tools is that this approach can (and should) be used before the application development process begin. By creating the Business Schemas first, the development of the application that follows benefits from the security-aware data access interfaces that they provide. Thus, development is faster, since there is less time spent writing and correcting CRUD expressions, and easier, since developers are no longer required to master the database schema and access control policies from the start. Furthermore, each time the application is compiled, any changes made to the access control policies are automatically reflected in the code and any errors that need correcting displayed when using an Integrated Development Environment (IDE).

2.1.2 Pre-Execution Phase

In the pre-execution phase, the application implements the Business Schema interfaces which are used during the execution phase to access the stored data.

First, the application uses the *Business Manager* to authenticate itself with the *Policy Server* on the server-side. Then, the application requests the Business Schema metadata that it is allowed to use (3). Once the metadata is received, the Business Manager implements the Business Schemas according to the architectural model (4), and any disparities between the implemented interfaces and the compile-time interfaces used by the application result in an exception.

Once all the Business Schemas are implemented and all match the existing interfaces, the application can move on to the execution phase.

2.1.3 Execution Phase

In the execution phase, the application uses the implemented Business Schemas to access and manipulate the stored data.

The application can request the *Business Manager* to instantiate a particular Business Schema, identified by the CRUD expression associated with it, and then use it to access the stored data.

As mentioned before, the Business Schemas mirror the standard data access APIs, allowing developers to quickly adapt to this architecture. For example, in the Java language, the Business Schemas mirror the Connection and ResultSet object APIs, two highly used database connectivity APIs.

However, the *Business Manager* includes a *Sequence Controller*, a module that controls the order in which Business Schemas are used. This is a previous iteration of the work presented in [31] and leverages the idea that CRUD expressions may be put together by malicious developers in ways that were not expected, allowing sensitive information to be leaked. By defining high-level use cases first, it is possible to determine which operations are required to be executed and in which order to perform

each use case. When a Business Schema is instantiated, the *Sequence Controller* checks to see if it is one of the available Business Schema according to the current sequence (5). Moreover, to make sure that the developers do not have to master the sequences, each Business Schema can instantiate the next Business Schema in the sequence.

Finally, once a Business Schema is instantiated, it can be used to query and explore the data or do something else, depending on the associated CRUD expression (6). To make sure that the data stored in the database on the server remains secure, the application never knows the database credentials. Thus, the *Policy Manager* functions as a proxy to connect to the database, meaning that the database credentials never leave the server.

2.2 Soft Requirements and Access Control Models

While access control models as a subject are fairly well known and studied early on in security-related courses, it is usually applied together with crisp logic to define very clear rules regarding what properties a subject must possess to be granted some permissions over the data. However, as previously argued some application scenarios may fall outside this context, possessing soft requirements that are not as easily defined as crisp access control rules.

There are many attempts in the literature at unifying these two concepts to create an access control model capable of handling soft requirements, usually through fuzzy sets and fuzzy logic theory. In this section, the most common approaches used to achieve this unification are introduced and discussed in terms of their benefits and drawbacks to provide some necessary background. For further reading on both of these topics individually, a summary of well-known access control models is provided in [18] and an introduction to fuzzy logic, sets and control systems in [40].

2.2.1 Classical Model Adaptation

One of the simplest approaches to incorporate soft requirements in access control is to replace the Boolean logic used in an existing access control model by a multi-valued logic such as fuzzy logic.

As an example, instead of having a set of users that belong to a particular role in a role-based access control model, the users have a degree of membership to each role as shown below instead.

$$USERS \times ROLES \longrightarrow [0, 1]$$

Then, depending on the degree of membership each user may be granted the associated permissions or not. This decision process may use some sort of threshold where the given role is granted to a user above a certain membership degree, or the user may retain the partial membership to each role. In the latter case, different permissions may require different minimum membership degrees to be granted to a user. An example of such a model is presented in [41] and partially in [42]. Both are further detailed in the state of the art.

While this approach has the benefit of being easy to produce and somewhat familiar to use given that it is based on existing access control models, it has the drawback of being restricted to the concepts

used in the base model. For example, in the case of an adapted role-based access control model, the soft requirements have to be based on the roles and nothing else. Another benefit of this approach is that auditing the access control policies is also easier to do, as the set of users is expected to be known. This means that the membership degrees can be read for each mapped user and their permissions evaluated.

Regardless, these types of models are an interesting stepping stone to learn about and research more complex fuzzy-based access control models.

2.2.2 Concept Specific Inference

Another very common approach to developing access control models based with support for soft requirements is to create a model for specific use cases. The biggest drawback of adapting an existing model is the restriction that it imposes on the possible concepts that can be used to define the access control rules, thus the idea of using an inference system based on carefully chosen concepts.

Many access control models follow this approach, such as the work presented in [23], [43] and [44]. These models take in specific parameters from the subjects requesting access to the data and sometimes even from other sources, such as the history of the interactions with that subject in past, the sensitivity of the data being requested, the severity of the action to be performed, etc. This information is used to calculate vague concepts such as the risk associated with an access attempt or the trust the system has on the subject, and the access requests can be granted or denied depending on membership degrees involved.

While this approach allows creating fuzzy-based access control models based on any concepts, each model focuses on just a few specific ones. Thus, an access control model created in this way is meant to be used in a very specific scenario, limiting its usability outside of it. Another drawback is that auditing the correctness of the rules is harder since generally there is no fixed set of users that are meant to access the system and any combination of input parameters may occur. However, these models are good when applied to the specific scenarios they were developed for and tend to be easier to fine-tune with expert knowledge.

2.2.3 Generalized Inference

The inference systems used for specific use cases are usually instances of more general fuzzy inference systems (FIS) where the rules, the input and the output variables have been specified. The more widely known and used FIS are the Mamdani-type FIS [45]; and the Sugeno-type FIS [46]. A quick comparison of these two system types follows.

While the input variables are treated the same way in both cases, the difference lies in how the output values are calculated. For instance, the Mamdani-type FIS can be used in multiple-input single/multiple-output systems, the Sugeno-type FIS can only be used in multiple-input single-output systems. This point alone makes the Mamdani-type FIS more appealing for access control as there can be multiple permissions that need to be evaluated. However, if the permissions are determined by a single factor, for example the level of risk, then the Sugeno-type FIS could be used as well.

Another difference lies with the fact that the Mamdani-type FIS has more expressive power and interpretable rule consequent than the Sugeno-type FIS. This is evident if the rules are compared side-by-side as shown in Listing 2.1.

LISTING 2.1: Example Mamdani-type and Sugeno-type rule comparison.

```
Mamdani: IF Service IS Excellent THEN Tip IS High
 Sugeno: IF Service IS Excellent THEN Tip IS F(Excellent)
```

While in the Mamdani-type FIS it is clear that the rule is meant to convey a high tip as a consequent, the Sugeno-type FIS loses this power by simply applying a crisp function. In an access control context, being able to interpret the rules is an important aspect to help ensure the that they are correct.

Finally, the Mamdani-type FIS has a non-continuous output surface while the Sugeno-type FIS has a continuous one. While this point means that the Sugeno-type FIS is better suited for mathematical analysis and systems that require a continuous output, for access control systems the Mamdani-type FIS is most of the time better as they usually only need to determine if a permission is granted or denied.

2.3 Summary

In this chapter, the previous architecture that was built upon in this work was introduced to provide background knowledge of its previous features and to help determine how it evolved. Furthermore, an explanation of how access control models are typically developed to incorporate soft requirements was provided to understand the major benefits and drawbacks of each approach. Finally, a quick overview of the differences between the Mamdani-type and Sugeno-type FIS including their benefits and drawbacks for application in access control contexts was also presented.

Chapter 3

State of the Art

In this chapter, the state of the art revolving around the various topics researched during this thesis and how the methods and systems presented in this dissertation enhance or complement it are carefully detailed and argued.

Some of these topics are included in the fields of access control and authentication/communication security. The former includes both crisp and soft-based access control, as well as the mechanisms that can be employed to protect the database schema from being disclosed. The latter focuses on both widespread technologies used to secure authentication schemes and communication protocols and solutions presented in the literature.

Therefore, this chapter is divided as follows: section 3.1 details the various crisp and soft access control models, as well as the various database schema protection mechanisms; section 3.2 details the various technologies, solutions and protocols used/proposed to provide secure authentication schemes and secure communication; and section 3.3 summarizes the findings of the chapter.

3.1 Crisp and Soft Access Control

Access control is the process of restricting access to a resource in a selective manner [1], usually by giving a subject proper authorization to be able to access said resource. There are many facets to access control that must be taken into consideration to make it secure, such as subject identification and authentication, encrypted communication channels, a correct access management system, etc.

In this section, several works related to these facets of securing access control is succinctly presented and discussed.

3.1.1 Crisp Access Control Models and Systems

In this section, a more detailed look into existing models, tools and related works that enable applications to access data stored in databases is presented.

Starting with tools that are widely available and used, Java Database Connectivity (JDBC) [2] is a Java-based technology that enables applications to access data stored in a database. As such, it is a prepackaged application programming interface (API) for the Java programming language that provides methods for querying, inserting, updating and deleting data in relational databases.

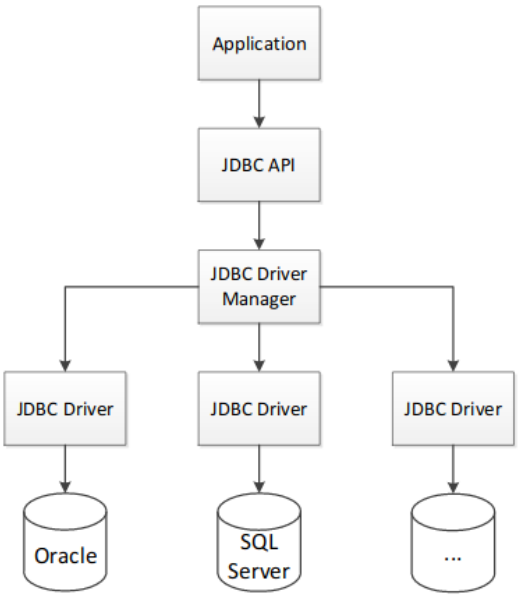


FIGURE 3.1: JDBC architecture.

As shown in Fig. 3.1, the application interacts with JDBC only through its API. For JDBC to handle multiple different databases, it uses a driver manager that selects the proper driver according to the database that it is connected to. Each driver manages the database-specific communication protocol, and new drivers can be dynamically loaded so they can be used.

JDBC allows applications to execute standard Structured Query Language (SQL) statements, such as *SELECT Name FROM People* which would select the name column value for every entry in the table *People*. The result of executing a query comes in the form of a table, with each row containing a single data entry that satisfies the query. To handle the result of query execution, JDBC uses the *ResultSet* object, which allows the application to iterate over each row in the query result.

ADO.NET [4] is another tool that comes prepackaged with the C# programming language and is equivalent to JDBC with just some different class naming conventions. Since these tools are more focused on being compatible with multiple relational databases rather than reflecting the access control policies set in place, attempting to access a table that the application does not have permission to access raises an error only while attempting to execute the query. Thus, to write an application using these tools the developers must master the database schema and what data they are allowed to access.

More sophisticated object-relational mapping tools exist that map the object-oriented domain model with the relational paradigm used in relational databases, such as Hibernate [5], Language Integrated Query (LINQ) [3], Java Persistence API [47] and EclipseLink [48]. These tools create objects that map to database tables (or vice-versa) and provide persistence engines that allow modifications made to the objects to be transparently carried over to the associated database tables. However, while the object creation process does take the access control policies into account (since objects are mapped

to tables the application is allowed to access) if the policies are modified at any point after the mapping has to be remade. This process not only needs to be triggered manually, but such changes to the access control policies are still only detected during execution through access permission errors.

Ur/Web

Chlipala et al. [49] created a tool, Ur/Web, which allows create, read, update, and delete (CRUD) expressions to be extended to check the access control policies in a system backed by a DBMS as they are executed.

Using this tool, programs can be developed and checked to ensure that the data that is retrieved and manipulated by the CRUD expressions are accessible through some policy. The extension allows CRUD expressions to be parameterized to capture the "secrets that user knows", thus allowing the same CRUD expressions to be used with different users.

Listing 3.1 shows how this extension accomplishes that by using the predicate *known*, which models what information the users are already in possession of. This information is used to decide what information can be disclosed to each user.

LISTING 3.1: Example Ur/Web CRUD expression.

```
policy sendClient {
    Select *
    From user
    Where known(user.pass)
}
```

In the listing example, the policy *sendClient* governs a CRUD expression that retrieves information about users. However, the user requesting this data receives the information only for users that he already knows the password of.

This approach allows to ensure that data flows according to the access control policies, but the *Where* clauses themselves are not checked. This may allow protected data to be leaked implicitly, for example by changing the *known* clause and comparing the results. Furthermore, the validation process occurs only at compile-time, so the developers still have to master the database schemas and the defined access control policies to write the applications.

Integrating access control policies within database development

Abramov *et al.* [50] present an approach where the security aspects are defined at the early stages of software development, instead of at the end as is commonly the case. To achieve this, a model is presented that infers and applies access control policies.

A new methodology is proposed that enables security patterns to be clearly defined using common modelling techniques and using modelling languages such as the unified modelling language and the object constraint language.

The methodology is divided into four major phases: preparation, analysis, design, and implementation. While the latter three occur at the application development level, the preparation phase is carried out at the organizational level.

The preparation phase concerns the security officers and domain experts, which specify the organizational security patterns and transformation rules. These patterns are later enforced during the development of any application and database schema, while the transformation rules specify how to transform the application model based on the patterns into database code. The analysis phase includes the creation of a conceptual data model, which is based on the user's requirements, and a functional model, which is based on functional requirements. This phase also includes the specification of the security constraints, expressed as a table that aggregates all access privileges. In the design phase, the models created during the analysis phase are transformed into a coherent model that adheres to the organizational model. This includes the refinement of the data model, where the initial class diagram and the access privileges specification are unified into a single class diagram, and conformance checking, which validates the unified class diagram against the specified security patterns. Finally, a developer at the implementation phase has access to a conceptual data model that is augmented with the security constraints and that is compliant with the organizational security patterns. The transformation rules can then be applied to implement the security specifications into the database code.

However, while this approach allows creating software that has the security aspects defined and incorporated early in the development life-cycle, the development of the application still requires the developers to master the database schema and access control policies to correctly write the necessary database queries.

Hippocratic Databases

Hippocratic databases are databases that are designed to incorporate privacy policies into their architecture, and they are defined by ten principles [51]:

- **Purpose specification**. Data collection operations must have associated the purpose for which the data is being collected;
- **Consent**. The donor of the data must consent the associated purpose;
- **Limited collection**. The amount of data collected must be the minimal amount that satisfies the purpose;
- **Limited use**. Only queries that are consistent with the associated purpose must be allowed to be executed;
- **Limited disclosure**. The data shall not be disclosed outside the database for any reason other than the consented purpose;
- **Limited retention**. The data collected can only be retained for the minimal amount of time necessary to satisfy the purpose;

- **Accuracy**. Data must be up-to-date and correct;
- **Safety**. Data must be protected against theft and unauthorized access by security mechanisms;
- **Openness**. A subject must always have access to all data it is the donor of;
- **Compliance**. The donor of some data must be able to verify the compliance of the principles.

These principles have been tested with PostgreSQL [52], and Listing 3.2 shows an example query that could be executed in that database system.

LISTING 3.2: Example Hippocratic PostgreSQL query.

```
Select s.saleNumber, s.saleValue, s.taxValue
    From Sales s
    Purpose auditing
    Recipient salesManager
```

The query shown in this listing produces a result that has its columns restricted for both the purpose and the recipient. Furthermore, only the data intended to be used for auditing is shared. Thus, it is clear that Hippocratic databases handle a more unusual aspect of access control, i.e. privacy, as it lets the donors of some data specify how, when and for what purpose it can be disclosed for.

LeFevre *et al.* [53] presents a method to limit the data disclosed in Hippocratic databases by employing the query rewriting technique. The policies are defined using either EPAL [54] or P3P [55], and it specifies who can access the data and for what purpose through rules. When a query is submitted and the database returns the result, the application processed the records and filters out any that contains prohibited information.

SESAME

SESAME [56] is a dynamic context-aware access control mechanism for pervasive GRID applications. Based on the user's context, it can dynamically grant and adapt permissions to complement existing authorization mechanisms.

To enable this approach, an extension to the classic Role-Based Access Control (RBAC) model is used, called dynamic RBAC. When users log in, it assigns to each user a default role hierarchy and then proceeds to monitor their context to assign roles as needed. The context used in this approach can be either an object context or a subject context. The object context contains information such as the user's location, time, local resources and link-state, while the subject context contains information such as the system's current load, connectivity to a resource and availability.

While this approach can assign roles dynamically to each user as needed, it does not change the application development flow like most works presented in this section.

SELINKS

SELINKS [57] extends the LINKS [58] programming language, which is a language similar to LINQ that can be used in the development of secure web applications.

A program written with LINKS is compiled to create the byte-code for each tier of the application alongside the security policies. User-defined functions on the RDBMS are then created, which encode these and check at runtime what actions each user is allowed to execute. Programmers can define security labels, i.e. types that define the metadata, that are used by functions that enforce the policies to mediate the access to the data. To ensure that the data is accessed only after the proper policy enforcement function is invoked, a type system called Fable [59] is used.

SELINKS improves upon LINKS by integrating a security context in every application tier, using Fable to ensure security policies are followed. This is optimized to reduce network load by carrying out the permissions check on the user-defined functions in the database, instead of transferring the data to the webserver to check. Furthermore, it is a single tool, lowering the toll on developers from having to learn multiple tools.

However, since the security labels a group-based access control policy that only distinguishes between read and write operations, it is not possible to apply the restrictions on the more abstract query level.

Jif

Jif [60] is a programming language that is security-typed and extends Java to provide support for information flow control and access control, both at compile-time and runtime. Java is extended through the use of labels that express the access control policies that are in effect and that should be enforced, as well as how the information may be used.

Listing 3.3 shows a variable declaration in Java extended with Jif, in which the x variable is declared as an integer alongside a security policy.

LISTING 3.3: Example Java variable declaration using Jif.

```
int {Alive → Bob} x;
```

In this scenario, the label expresses that the information stored in x is controlled by the principal Alice and that the information may be accessed by the principal Bob. Therefore, the Jif compiler is capable of analysing the information flow within programs to ensure that the policies expressed as labels are followed. Alongside labels and principals, Jif also supports principal hierarchies, integrity and confidentiality constraints, authority delegation between principals, confidentiality, integrity downgrade and a form of label polymorphism.

However, while this language manages the information flow at the application level, other tools and mechanisms must be used to manage the data access to databases.

Reflective Database Access Control

Olson *et al.* [61] presents the Reflective Database Access Control. Instead of defining user permissions for each table through access control lists, this model expresses privileges as database queries. This approach has the benefit of making permissions depend on data elsewhere on the database, and not

be statically defined. The reflective access control policies are expressed using the Transaction Datalog [62] to provide formalism, and implementation of this access control model was presented in [63].

While this model takes a very similar approach to the Secure, Dynamic and Distributed Role-based Access Control Architecture (S-DRACA), it still lacks many of the additional features, such as the client-side data access interface generation and implementation, parameter flow protection, query sequencing enforcement, etc. Furthermore, the permissions are bound to the querying language used by the database. If the database needs to be replaced, the permissions have to be redefined. S-DRACA also suffers from this issue, as permissions are statically defined as SQL CRUD expressions, and is tackled by the work presented in this dissertation.

Security-driven Model-based Dynamic Adaptation

Morin *et al.* [64] presents a security-driven model-based dynamic adaptation to address an issue where even if the separation between the policies and the application code is done in theory, the reality is that it is never fully accomplished in practice. The consequence of this problem is that some of the policies end up being expressed directly in the application code.

The approach presented uses meta-models, which defines both the access control policies and application architecture. It also defines how to statically and dynamically map from the access control policies meta-model to the application architecture meta-model.

However, that is the entire scope of the presented work. Thus, it does not implement secure and dynamic security mechanisms for data access.

Java EE

Java Enterprise Edition [65] is an extension of the standard edition of Java used to build enterprise software. In this edition of Java, annotations can be used to enforce RBAC policies directly in the application code at the method level. These annotations allow developers to declare roles and to specify which of those roles are allowed to invoke certain methods.

Listing 3.4 shows how this can be done at a basic level. The roles *Administrator*, *Manager*, and *Employee* are declared in the *Product* class using the *@DeclareRoles* annotation. The *setDiscount()* function is declared in this class, which is annotated with *@RolesAllowed*. This annotation states that only users with the *Administrator* role are allowed to invoke this method.

LISTING 3.4: Java EE roles example.

```
@DeclareRoles({"Administrator", "Manager", "Employee"})
public class Product {

    @RolesAllowed("Administrator")
    public void setDiscount(double price) {
      ...
    }
}
```

However, the specific users that invoke these annotated methods are not identified, meaning that anyone who has one of the allowed roles has access to the protected method. Furthermore, this approach only checks if a user is allowed to execute a method at runtime, meaning that developers have no way to validate that their code adheres to the access control policies statically.

Annotated Objects

Fischer *et al.* presents in [66] Object-sensitive RBAC, an extension of RBAC that can be used with object-oriented programming languages. It attempts to address some of the shortcomings in the current RBAC model and associated frameworks such as the Java Enterprise Edition discussed in this chapter.

Listing 3.5 shows some sample code using the standard RBAC in Java Enterprise Edition. The intent is that the doctor of a patient and the patient himself should be allowed to access its data. However, any user with the role of *Doctor* or *Patient* is allowed to invoke the method. Thus, it is clear that the access control policy cannot be implemented as intended.

LISTING 3.5: Standard RBAC example.

```
public class Patient {
    private int patientId;

    @RolesAllowed({"Doctor", "Patient"})
    public static Patient getPatient(int pid) {
      ...
    }
}
```

Object-oriented RBAC addresses this shortcoming as shown in Listing 3.6. In this scenario, the roles *DoctorOf* and *Patient* are used, and both of them are parameterized by a patient identifier as shown in the *@Requires* annotation. Thus, when the method is invoked with a specific patient identifier, the user must be the doctor of a patient with that identifier or be that patient.

LISTING 3.6: Object-oriented RBAC example.

```
public class Patient {
    @RoleParam public final int patientId;

    @Requires(roles={"DoctorOf", "Patient"}, params={"pid", "pid"})
    @Returns(roleparams="patientId", vals="pid")
    public static Patient getPatient(@RoleParam final int pid) {
      ...
    }
}
```

This work also possesses a type system that allows developers to write access code knowing if they are violating any access control policy or not.

In a complementary work, Zarnett *et al.* [67] presents a method to control access to methods of remote objects via Java remote method invocation [68], which allows an application to use objects that exist in a different application, possibly running in a different machine. By enriching the objects with metadata about the roles that are authorized to use them through annotations, the proxy objects that handle the requests to execute methods can be generated following the access control policies. Since the proxy objects are tailored to each user with only the method they are allowed to execute, users cannot attempt to execute methods that they are not allowed to.

However, none of these solutions eases the writing of queries to the database as developers must still master the database schema and associated access control policies to do so.

Predicated Grants

Chaudhuri *et al.* [69] proposed adding predicates to grants to achieve a fine-grained authorization model. The approach allows defining which records a user can access within a table, what the public can see, etc.

Listing 3.7 shows a simple example of a query using this approach, where each employee can access their employee information and all other data is nullified so it cannot be read.

LISTING 3.7: SQL example with predicated grants.

```
grant select on Employee
    where (employeeID=userID())
    else nullify to public
```

An advantage of this model is that it addresses cell-level security by nullifying values. It also enables predicates to be used in any kind of grant, such as CRUD expressions, stored procedures and functions. Furthermore, aggregation functions can also be authorized while restricting access to the underlying data and it also has mechanisms to handle large numbers of users can database objects.

While this model does enhance the privacy of the data by ensuring that data is only disclosed to a user if it satisfies the predicated grants, it does not help the software developers to write the data access queries without having to master the schema and the access control policies.

λDB

Caires *et al.* presented in [70] a programming language for data-centric programs that can enforce data access control policies through static typing. It uses data structures known as *entities* that are checked against the access control policies and another context-dependent information at compile-time. Permissions are associated with *entities*, which are comprised of: the action granted (i.e. either read or write), the attributes of the entity, and a logic condition. Listing 3.8 shows an example of such an entity.

LISTING 3.8: λDB permission example.

```
entity Person [userid:string;public:string;photo:picture;secret:string]
    ...
    read public where true;
    read secret where Auth(uid) and uid = userid;
    read photo where Auth(uid) and Friends(userid,uid);
    write where Auth(userid);
```

This entity is named *Person* and is defined by four attributes: *userid*, *public*, *photo*, and *secret*. The conditions shown define that the *public* attribute can be always read, the *secret* attribute can be read-only by its owner, and that the *photo* attribute can be read by its owner and its friends. The condition for the write permission applies to all attributes and only allows the owner to update the attributes.

This approach provides only a single action capable of authorizing update, insert and delete operations (i.e. the write permission) on the attributes, so it is not possible to differentiate them. However, the *where* clauses can be protected in contrast to previously shown solutions, such as Ur/Web.

Assurance Management Framework

A solution similar to the approach used in this work is presented by Ahn *et al.* in [71] where a tool that can generate some source code from a security model is defined in order to validate it. Therefore, this generated code can be used to check if the model and policies violate consistency or validity. It does so with four tasks:

1. **Model representation.** The security model is represented in the Unified Modeling Language.
2. **Policy specification.** Perform visual and logic-based policy specification, which is then translated into an high-level policy specification to be integrated into the system design.
3. **Model and policy validation.** The security models and policies are checked in terms of their consistency and validity through a set of system states applied against them.
4. **Conflict detection and resolution.** Resolve any conflicts found between policies to ensure that a policy does not conflict with other existing policies.

This framework is concerned with validating the policies that the applications are meant to follow for data access and not how the developers should write the application code that does so. This way, the developers still need to master the database schema and to write the database queries themselves. This thesis aims to complement frameworks such as this one, by generating the source code that accesses the database from the access control policies.

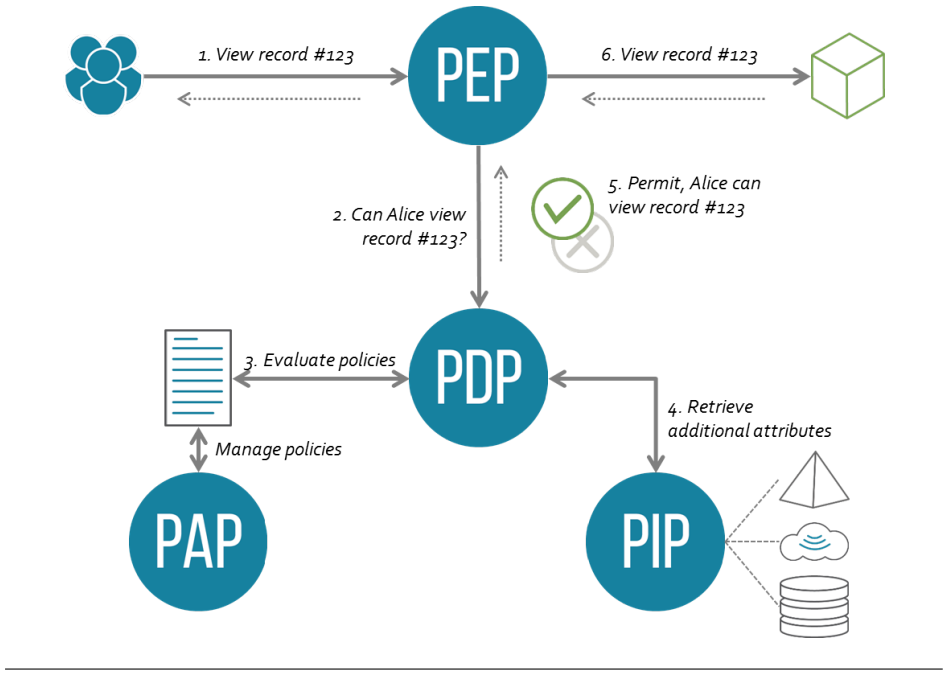


FIGURE 3.2: XACML architecture and sample authorization flow.

XACML

The extensible access control markup language (XACML) [72] is a declarative access control policy language that is implemented using extensible markup language (XML) and a processing model that describes how requests should be evaluated according to the rules defined in the policies. It uses the model shown in Fig. 3.2 to enforce access control policies.

When a user issues a data access request, the Policy Enforcement Point (PEP) intercepts it and communicates with the Policy Decision Point (PDP) to check whether or not the user is authorized to do so. The PDP then requests the policies to the Policy Retrieval Point (PRP) (not shown in the figure) and any necessary additional information from the Policy Information Point (PIP). Once the PDP determines if the user is allowed to access the requested data, the decision is sent back to the PEP which enforces the decision. When the access request is granted, the PEP uses some static business logic to handle the request. The Policy Administration Point (PAP) is used for administration purposes such as managing the access control policies. However, changes made to the access control policies are not reflected automatically in the PEP logic, so it must be updated in advance.

Other Works

Others works related to access control enforcement include a new technique and a tool called Mohawk that can detect errors in the RBAC policies through abstraction refinement proposed by Jayaraman *et al.* [73]. This tool adds roles to the abstraction in refinement steps in order to find errors. Wallach *et al.* also proposes new semantics for stack inspection that aims to solve issues with the traditional stack

inspection [74], such as the detection of dangerous system calls (e.g. to the file system) or verify if it is allowed.

The work presented in this dissertation aims to complement these, as there is no mechanism in place yet to check for errors in the actual data access policies or the usage of dangerous methods in the application code that is not related to data access. It is important to check that the access control policies defined for S-DiSACA are correct, as the application code generated to access the data is completely dependent on them.

3.1.2 Soft Access Control Models and Systems

The fuzzy set theory is a topic that has been researched in recent years to tackle scenarios where the information that needs to be processed is vague, which can include operations research, management science, politics, social psychology, artificial intelligence, and access control, among others [75]. While there are some applications of fuzzy set theory in access control systems, which are shown in this chapter, no research was found around their correctness auditability for or the security challenges that arise from the application of fuzzy logic.

Adaptive Risk-based Access Control

Atlam *et al.* presents in [44] an adaptive risk-based access control model for the Internet of Things (IoT) based on fuzzy logic and expert judgment. This model estimates the security risk associated with each access request and as such tackles the problem of optimizing the risk estimation techniques. It is also stated that the goal of IoT is to increase information sharing and that an access control system must also be auditable. To this end, the model accepts four different inputs: user context; resource sensitivity; action severity; and risk history.

User context contains real-time features that represent the user/agent with contextual attributes while it makes the access request, such as location and time. Different user context has distinct risk values.

Resource/data sensitivity conveys the level of importance of the data, which affects how much risk associated with an access request is tolerated. As the authors point out, mapping data sensitivity levels to data is a subjective process that depends on how valuable the data is to its owner. To simplify this process, the solution of using security experts to categorize the data is proposed.

Action severity indicates what is the impact of the action that the user/agent wants to perform over the data with its access request. Security experts can categorize the available actions and map them to a severity metric. This way, a risk metric is applied to each action over a specific resource.

The user risk history is a history of the risk values associated with the previous access requests made by a particular user. This is used to monitor the user's behaviour patterns and to differentiate good users from malicious users.

A risk estimation module takes these inputs and uses them to estimate the overall risk value related to the access request. The risk value is then compared within the risk policies to make access control decisions. The risk policies are defined with thresholds that determine that an access request should be granted if the risk value is lower, or should be denied if it is greater.

The authors also define auditability as the process of collecting evidence of the various access operations performed by each user. While this is enough for access control models where the permissions associated with each user are explicit in its policies, the concept of auditability should be expanded once anyone is allowed to issue access requests and the access control decisions are performed using fuzzy logic on a request by request basis. This is especially true if there are no security experts available. Thus, it becomes important to understand under which conditions do user requests get accepted to determine the correctness of the policies. Other security issues that should be considered when using this model are also not discussed.

Fuzzy Role-based Access Control

In [41], the authors introduce Fuzzy Role-based Access Control, which uses fuzzy relations between users-roles and roles-permissions. The authors use the following notations and definitions:

- $USERS$ is a set of users.
- $ROLES$ is a set of roles.
- OBS is a set of resources (objects).
- OPS is a set of operations.
- $PRMS = 2^{OBS \times OPS}$ is a set of permissions.
- $UA \subseteq USERS \times ROLES$ is a set of user-role assignments.
- $PA \subseteq PRMS \times ROLES$ is a set of role-permission assignments.

These notations and definitions are then used to define the user-role and role-permission assignments, which forms the core of this model:

- $UA : USERS \times ROLES \longrightarrow [0,1]$
- $PA : ROLES \times PRMS \longrightarrow [0,1]$

This approach allows for users to have partial permission assignments, which are then used to calculate the access degree they have to resources. Then, if the access degree is used directly to control access to a resource, the resource itself must have fractional access, defined using the following access function:

- $access : USERS \times OPS \times OBS \longrightarrow [0,1]$

When access cannot be fractional, i.e. it must be either granted or denied, we enter the context of this thesis which aims to make the output a binary access control decision. In such cases, the authors define a function that takes a threshold variable δ (i.e. a value between 0 and 1) and returns "grant" if the access degree is greater than δ or "deny" if not.

In summary, the authors adapted the classic role-based access control model, which requires an explicit mapping between users and their roles to exist before any access attempt. This makes auditing this model trivial, as the membership degrees can be read for each user and a determination of which permissions they are granted or denied can be made. However, in the case of systems where such mappings between users and permissions are made in runtime according to user parameters, this is not possible. Previous users can be easily audited but their parameters may change between requests, and new users may request access at any time. Thus, from a security standpoint, it becomes important to discover which input parameter ranges grant permissions and have an expert determine their correctness. This thesis aims to complement this work in this regard.

Fuzzy Trust-based Access Control

In [23] a work was proposed by Mahalle *et al.* in which trust level of devices is quantified, so a fuzzy approach to trust-based access control could be achieved. This is done by capturing information about the devices to determine the vague concepts Experience (EX), Knowledge (KN) and Recommendation (RC), and several fuzzy sets (linguistic terms) were defined for each one. The following values for each concept are calculated for a particular context c between two devices A and B and then used as inputs for the membership functions of the linguistic terms.

EX depends on the history of interactions v_k between A and B, where $k \in [0, n]$, incrementing every time a positive interaction occurs and decrements otherwise:

$$(EX)^c = \frac{\sum_{k=1}^{n} v_k}{\sum_{k=1}^{n} |v_k|}$$

KN is calculated with the help of direct knowledge (d), indirect knowledge (r), and their respective weights (W_d, W_r), where $d, r \in [-1, 1]$, $W_d, W_r \in [0, 1]$, and $W_d + W_r = 1$:

$$(KN)^c = W_d * d + W_r * r$$

The RC is calculated by device A based on the summation of the RC values from n other devices about device B. W_i and $(r_c)_i$ are weights assigned by device A to the recommendation of i^{th} device and the RC value of i^{th} device respectively, where $r_c \in [-1, 1]$ and $W_i \in [0, 1]$:

$$(RC)^c = \frac{\sum_{1}^{n} W_i * (r_c)_i}{\sum_{1}^{n} (r_c)_i}$$

These fuzzy sets are then used to determine the level of trust that a user or another device can have to that device. Different permissions can be mapped to different levels of trust, so depending on the level of trust the granted permissions change. This is achieved by defining an ordered set with access rights, where its cardinality is equal to the number of trust levels, then a specific trust level has specific access rights. While this is a valid approach, it makes access decisions solely based on the level of trust. If there are other access conditions, they need to be considered separately as the system is only built to handle the concept of trust. Furthermore, no discussion over the various security aspects associated with an access control system is made.

TABLE 3.1: Fuzzy sets used in the input fuzzification step.

Input	Fuzzy Sets
Data Sensitivity	Low Medium High
Action Severity	Not Sensitive Sensitive Highly Sensitive
Risk History	Low Moderate High

Fuzzy Risk-based Access Control

Another work was carried out by Li *et al.* that uses fuzzy set theory to calculate a measure of risk and applies it to enhance the access security of eHealth cloud applications [43]. This fuzzy set theory application comes from the "urgent need for effective access control to protect highly sensitive healthcare information over a cloud computing environment." While other approaches are mentioned by the authors that incorporate risk management into the access control decision making, they argue that these approaches have the drawback of using different factors to estimate risks and that the risk levels are mainly qualitative. Fuzzy set theory is an alternative technique to address the uncertainty and the qualitative nature of the risk levels during the risk assessment.

To achieve this, three different inputs are used: **data sensitivity**; **action severity**; and **risk history**. These inputs are fuzzified into three different fuzzy sets each, as shown in Table 3.1.

Next, a set of rules is applied to calculate the level of risk associated, which can apply to five different fuzzy sets that define the risk levels: negligible, low, moderate, high, and unacceptable high. These rules are determined by experts.

A crisp output value is then determined by applying a defuzzification technique, which indicates the overall level of risk as a percentage. However, the process to determine whether the access should be granted given a risk level is not detailed. Furthermore, correctness auditability is not mentioned in the paper, which could be a deterring factor, especially when dealing with healthcare applications. The security risks related to the application of fuzzy logic in access control scenarios should be considered.

Fine-grained Data Access Control with Attribute-hiding Policy

In [76], Hao *et al.* address a security issue with ciphertext-policy attribute-based encryption (CP-ABE), which is an approach that provides fine-grained access control to data in IoT, such as data exported to the cloud. CP-ABE achieves this by enabling data owners to encrypt their data under certain access control policies over a set of attributes. Then, the recipients of the data are allowed to decrypt the data if their attributes satisfy the access control policy associated with the ciphertext.

The issue with this approach lies with the fact that access policies are usually explicitly appended to the ciphertext, which enables anyone who obtains the ciphertext to be able to potentially infer some

information about the contents of the data or who are the recipients from the policy. This results in the disclosure of the underlying ciphertext and potential recipients.

To handle this problem, the authors propose hiding the whole attributes from the policies. This is achieved by removing the mapping function ρ from the linear secret sharing scheme-based access policy (M, ρ), which effectively hides the attribute information. However, this is not enough. While legitimate recipients can query if an attribute is in a policy through a Bloom filter, this approach is still susceptible to dictionary attacks. Thus, the authors propose using a fuzzy attribute positioning mechanism based on a garbled Bloom filter. In this approach, legitimate recipients can query the row numbers for their attributes and are capable of verifying the results by successful decryption while unauthorized recipients are unable to compromise the confidentiality of any valuable attribute.

This work focuses on a particular security issue related to data that is outsourced to the cloud and how that data can be kept private while providing access to legitimate recipients. While the data can be encrypted, the access control policies may disclose some information about the data or who the intended recipients are. The model designed in this dissertation has a somewhat similar approach in the sense that the mapping function between the recipients and the data is not explicitly defined in each policy. Instead, the model uses fuzzy inference systems to infer whether or not a recipient is to be given certain permissions over the data or not based on their attributes.

One key difference is that the policies used in [76] use crisp attributes in their rules, which aims to provide a more fine-grained selection of the recipients of the data. The model proposed in this dissertation aims to be more flexible by determining the membership degree of each user requesting access to the data to a set of fuzzy sets, and the access control decisions are then made based on fuzzy rules. Thus, it is intended to be used in situations where defining crisp access control policies is not feasible, such as preventing users with a recent history of vandalism from having write permissions on Wikipedia pages.

However, the possibility exists that if unauthorized users can get access to the access control policies, they could potentially learn which attributes to manipulate to obtain the permissions they desire. Thus, protecting access control policies is an important step to raise the security of the system using this model.

A Fuzzy Logic Based Trust-ABAC Model

An access control model based on the Attribute-Based Access Control (ABAC) model was proposed in [77] by Ouechtati *et al.* to address security and confidentiality issues stemming from the fact that a large number of life devices are integrated into heterogeneous networks (IoT).

When it comes to accessing these devices, it is difficult to ascertain whether or not objects are honest or malicious. Thus, the authors leverage the recommendations and social relations of users, which can effectively deal with some types of malicious behaviour that aims to deceive other nodes. In particular, the authors aim to address specifically the collusion attack, an attack where several objects can cooperate to increase or decrease their level of trust artificially.

This is achieved in two ways: by evaluating the trust level associated with recommendation messages; and by detecting collusion attacks to filter out the inappropriate recommendations.

The evaluation of the recommendations is performed based on the communities to which the object belongs since objects in the same communities have stronger links or have common affinities. This separation also allows to determine the similarity between recommendations to identify and limit the sources of attacks. Thus, recommendations are divided as follows:

1. The OOR recommendations, which are recommendations that come from objects that share the same ownership relationship with the recommended object.
2. The C-LOR recommendations, which are recommendations that come from objects that are physically co-located with the recommended object.
3. The C-WOR recommendations, which are recommendations that come from objects that meet the recommended object in the owner's workplace.
4. The SOR recommendations, which are recommendations that come from objects that meet the recommended object frequently, sporadically or continuously.

From this, the inputs *Internal Similarity* and *External Similarity* are calculated from which the output *Recommendation Value Credibility* is obtained using fuzzy logic. Then, the *Recommendation Value Credibility* is used again as input, alongside the *Degree of Social Relationship*, to obtain the *Trust Level*.

Compared to other approaches shown in this chapter, this work does not deal with many input variables. This allows the authors to show diagrams with the correlation between the inputs and the output for all possible values, allowing to easily verify if the model is correct or not. However, this simplicity means that this model is specialized in calculating trust levels, which is then be used in an ABAC-based access control system. It is also not possible to use custom vague concepts to make access control decisions with. Thus, while it is a promising approach in some IoT scenarios, it cannot process other variables that may be required to take into account during the decision-making process in other scenarios.

Context-aware Access Control with Imprecise Context Characterization

Context information can also be of high importance when making access control decisions. In [42], Kayes *et al.* argue that while some contextual information can be derived from crisp sets (such as the co-location of a patient and a nurse trying to access some of the medical records belonging to that patient), equally important contextual information cannot, such as how critical the health condition of the patient is.

To address this issue, the authors propose a *Context-Aware Access Control using Fuzzy Logic (FCAAC)* approach. To do so, four different stages are required in FCAAC:

1. Capture Low-level Data.
2. Derive Conditions.
3. Mapping Multiple Sources.
4. Make Access Decision.

In the first stage, the low-level contextual facts are captured from relevant context sources. These facts are then used in the second stage, where the relevant contextual conditions (both fuzzy and crisp) are derived from. In the third stage, all local data sources are unified to use the same data scheme. Finally, in the fourth stage, all the relevant contextual conditions are used to make context-sensitive access control decisions.

Looking at the FCAAC policy model, defined below, it is possible to see that it is denoted by a 4-tuple relation.

$$FCAAC = \langle U, R, CC, P \rangle$$

In this relation, U represents the set of system users that can request resources, R represents a set of roles, CC represents a set of contextual conditions, and P represents the set of permissions that allows the users to perform some operations over the requested resources.

This work aims to remain as close to traditional and more accepted access control models, namely RBAC, given how roles are used to denote a policy. Furthermore, there is a set of users that can potentially request access to the resources. The objective is to allow for contextual information to alter which permissions are made available to a user at a given point in time, for example, so an emergency doctor can get access to a patient's health information if the patient enters the emergency room in critical condition. The fact that the patient is in critical condition is a contextual piece of information that alters the data access needs of an emergency doctor.

This thesis has a broader scope and aims to provide an access control model that can be applied in situations where the set of users is constantly growing so that the data can be protected while being made available to any user that wishes to access it. While the objective of both models is quite distinct, both can use contextual information to alter the access control decision to a given user, so they can complement each other in this way.

Machine-learning Fuzzy-based Access Control

There are other approaches to build fuzzy systems capable of making decisions. One such approach is fuzzy decision trees [78], which were introduced in an attempt to adapt decision trees, one of the most popular methods used for learning and reasoning, to support and deal with uncertainties. This type of model, and others like it, could be used to make decisions once trained, however, a training dataset must be available. Furthermore, if the system learns continuously from the data it processes, then it is no longer deterministic since the decisions made can change over time for the same set of input values.

This is a problem because it becomes much harder to determine which input values are granted access to the resources at a given time. Even if this is not the case, there is no proposed method to verify that the access control decisions are correct for every possible combination of input values and that no outlier exists. Continuous learning only compounds this problem, requiring constant monitoring and auditing to ensure that the correctness of the system is maintained in the event the model is poisoned by learning from incorrect data.

Soft Systems Optimization

In [79], the authors apply an algorithm to optimize the fuzzy output function to improve the performance of rule-based fuzzy routing algorithms in wireless sensor networks. An aspect that has been lacking from every soft access control system discussed so far is the correctness auditing of the policies. This is partially caused by the complexity of the task, which would have to provide a security expert with the set of inputs that grant or deny each permission. These points are explained at length in section 5.2.

The approach of optimizing the output functions to improve performance has merit, as a similar approach was used during the development of the Binary Decision Fuzzy Inference System (BDFIS) to make it better suited for access control contexts. However, optimizing the policy auditing procedure requires further work and a careful analysis of the BDFIS as a whole. This attempt at analysing the whole system to try to predict binary output decisions has not been found in related literature.

3.1.3 Database Schema Protection

This section presents the different approaches used to secure the database schema. Regarding database schema, a lot of effort has been put into mapping the schema (the relational model) to the object-oriented paradigm usually present in applications [14], [80], [81]. For this, there are solutions such as Hibernate [5] and Eclipse Link [82], and even object-oriented databases [83]. They aim at freeing developers from the need to master the database schema, but they do not directly protect the schema used in the database. Programmers can still write CRUD expressions and evaluate the results of their execution.

A possible reason for the lack of effort put into protecting the database schemas is because there is already a commonplace solution present in most database management systems, which are the stored procedures [84]–[86]. Stored procedures do protect the database schema by encapsulating the CRUD expressions in the server-side. However, they also have some problems. Among them, the use of stored procedures does not scale well because in complex database applications, since the number of stored procedures would increase (to some degree) with the number of CRUD expressions. Another issue, potentially the most relevant, is that their names are static, meaning that they cannot be randomized. Due to this, users can try to execute them once they know their names.

There is also the possibility of using views to access the data on a database [69], [87], [88]. Views are defined with a select expression and users access the data provided by that expression, instead of accessing the tables directly. Nevertheless, the use of views does not scale well, their number also increases as the number of CRUD expressions increases, and are also usually only supported by relational databases.

One argument is that usually client applications do not access the database directly anymore since a multi-tier architecture allows the application processing and data management to be physically separated. While this scheme does not allow a client to connect directly to the database, the server the client application connects to does. Given that the server can be the target of malicious attacks, simply storing the data store queries in them is still not secure. Furthermore, developers for the server

application can still write and execute CRUD expressions, thus raising the chance of a successful internal attack.

Finally, in [15], [27] is presented an architecture where business logic is dynamically built at runtime and per the established access control policies. Thus, CRUD expressions are deployed in each application but only at runtime. Nevertheless, malicious users can resort to reflection mechanisms to disclose the database schema. S-DRACA, which was built on the concepts presented in part by these works, resolved this issue when applied to relational databases [34]. However, it relies on features that do not exist in every data store.

3.2 Authentication and Communication Security

This section introduces various standard solutions in regards to authentication and communication security, as well as solutions proposed in published works. The objective is to show that while the standard solutions fill the need for the majority of scenarios, there are some assumptions made that do not always hold up. Furthermore, programming languages and tools do not implement every proposed authentication and data encryption schemes.

3.2.1 Transport Layer Security

The Transport Layer Security (TLS) [89], [90] is an evolution of the now deprecated Secure Sockets Layer (SSL) [91], which is a cryptographic protocol created to provide security to communications in a computer network.

The primary goal of this protocol is to provide secure communication channels between two peers, such as a web browser and a web server, and such a channel provides the following properties:

- Authentication: The server should always be authenticated, while the client can be optional. This authentication can be performed using a variety of secure algorithms;
- Confidentiality: The data that is sent between two peers connected using a TLS channel should only be visible to them;
- Integrity: Data sent over an established TLS channel cannot be modified by a malicious entity without being detectable.

Due to these properties, this protocol has seen widespread use to secure everyday actions such as web browsing, emailing, instant messaging and voice calls over the internet (i.e. voice over IP).

However, many of these applications of TLS utilize an algorithm based on digital certificates to authenticate the servers and to establish a secure communication channel. Digital certificates can certify that an entity is whom they claim to be by using asymmetric encryption. Asymmetric encryption relies on two keys, a public and a private key, and data encrypted using one can only be decrypted by the other. So, if a server provides its digital certificate (which carries the public key) and encrypts a piece of information that the client knows, the client can use the provided public key to decrypt it and validate it to authenticate the server.

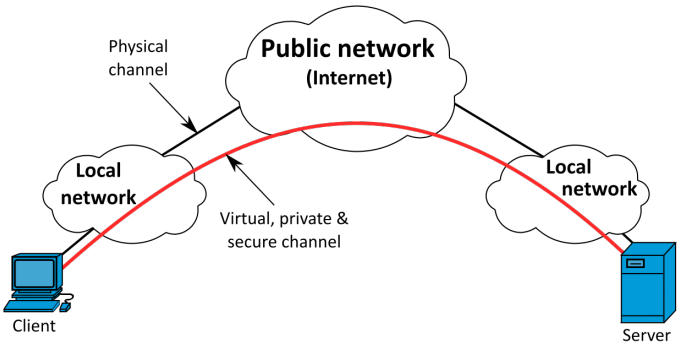


FIGURE 3.3: VPN connectivity overview.

The issue is that any malicious entity could potentially create their own certificates stating that they are the server. To prevent this from being possible, a public key infrastructure [92] was designed that maintains trust anchors known as Certificate Authorities, amongst other features. These can verify the identity of a server before issuing their digital certificates, which are now signed by the trust anchor. The idea is that the certificates of these trust anchors come pre-installed in the client devices, allowing to validate the authenticity of a server's certificate, and a version of JDBC with integrated SSL has been proposed in the past [93].

The problem comes from scenarios where trusting a Certificate Authority to not issue certificates to servers that are not whom they claim to be is not acceptable. Indeed, Certificate Authorities have existed that did that, either by mistake or malicious intent [94], and other issues can also be raised [95]. In these cases, attempting to use different methods of establishing TLS channels can be frustrating, as programming languages such as Java rely solely on digital certificates. Furthermore, requesting a certificate to be signed by a Certificate Authority can also be a costly venture. One of the goals of this work is to address this situation.

3.2.2 Virtual Private Network

Virtual Private Networks (VPN) [96] are used to extend private networks across public networks such as the internet, allowing users to access data and services as if they were connected directly to the private network where they reside, as shown in Fig. 3.3.

Since a VPN allows remote users to access private networks and potentially confidential data can flow through it, the connection is often encrypted and both the VPN server and the user are authenticated to ensure that the data remains private and is disclosed only to allowed users. Therefore, using a VPN to allow applications to access data stored in a database directly is a possible solution.

However, there are several problems with using a VPN for this purpose. A private network may have other services running alongside a database, and if other users are given access to the network through the VPN, they may attempt to access the database. This naturally reduces the security of the solution. A login layer could be applied on top of the database, but in this situation, the database access application needs to store these credentials, which could potentially be stolen and used in an

internal attack. If the database service is served by a VPN, then similarly to the previous login layer solution the database is vulnerable to internal attacks. Furthermore, if the VPN server is compromised, then the database becomes much more exposed to attacks, and if looking at the application code discloses the credentials used to access the database, then the data becomes accessible.

A solution capable of securing the database credentials outside the database access applications is preferable. Solutions that connect directly to the database cannot be secured for remote access using a VPN given the reasons above, which includes the previous work S-DRACA. While the S-DRACA utilizes a proxy server to not store the database credentials on the database access application, they could still be disclosed in an internal attack. This is a security aspect analysed and improved with this thesis.

3.2.3 Other Authentication Protocols

Other authentication protocols have been proposed and standardized, such as Kerberos [97] and the Remote Authentication Dial-In User Service (RADIUS) [98].

Kerberos authenticates users using an authentication server, separate from the server a user may want to connect to use some service. Once users authenticate with the authentication server, they receive a *Ticket-Granting-Ticket*. This ticket can be used on a ticket-granting server to issue *Client-To-Server* tickets to the services the user wishes to access. These tickets can then be used on the service servers themselves to authenticate the user himself and the server, after which the user can start issuing requests.

RADIUS is a client/server protocol that provides Authentication, Authorization and Accounting management for users. On the one hand, the clients are responsible for providing the RADIUS servers with user information and to process returned responses. The servers, on the other hand, are responsible for handling user connection requests, authenticating the user, and all configuration information required so the user can access the service requested. Communication between clients and servers are encrypted using a shared secret, and several authentication mechanisms are supported. However, a note in [98] warns that data can be lost and performance can degrade when the protocol is deployed in large scale systems.

While these protocols can be used to authenticate users to use a protected service [99], their application in the S-DRACA would not prevent the proxy server from still having to store the database credentials.

3.2.4 SSL/TLS Session-aware User Authentication

In [100], Oppliger *et al.* introduces a session aware user authentication to enrich SSL/TLS communication channels with a technique to prevent Man-In-The-Middle attacks from being carried out. A Man-In-The-Middle attack, as the name implies, consists of an attacker intercepting the communication between two entities and relaying it between them. This scenario has the effect of destroying any confidentiality benefit to be gained from encrypting the communication, as the attacker impersonates each entity and can read the data while sending it back and forth.

As previously mentioned, SSL/TLS communication channels use digital certificates to authenticate the server and optionally the client, thus preventing this kind of attack. However, Oppliger *et al.* argues that this is only true if both entities perform the authentication correctly. If a user does not check that he is connected to the correct server (and was not redirected to a similar server posing as the correct one with its own digital certificate), then the attack is still possible. This is especially true since the user is seldom authenticated by the server.

The proposed idea is twofold: to enable user authentication while coupling its secret credentials to the SSL/TLS connection state, which is different between connections. Thus, since in a Man-In-The-Middle attack scenario there are two SSL/TLS communication channels used (user $\leftrightarrow$ attacker and attacker $\leftrightarrow$ server), if the user authentication depends on the connection state then the server is able to determine that the user credentials were not sent on the same communication channel. The server can then conclude that an attack is likely taking place on the connection. This work was later revisited [101] to introduce further improvements.

While this approach does authenticate the users and introduces additional safeguards against attackers, it does not address the issues of internal attacks services running on the server (such as a database), in which case an internal direct connection is enough to allow a malicious user to access the data if the credentials are obtainable from the client application.

3.2.5 Multi-context TLS

In [102], Naylor *et al.* propose the multi-context TLS, an extension to the TLS protocol to support the middleboxes to be introduced in the middle of TLS connections. While the work presented in section 3.2.4 aimed to prevent Man-In-The-Middle attacks, this work aims to leverage of ability to be able to read data in the communication channel by purposely introducing entities across the connection.

The author argues that while the current TLS protocol provides entity authentication, data confidentiality and integrity, it assumes that all functionally reside on the endpoints. In reality, middleboxes are used along the path to provide a variety of services such as intrusion detection, caching, parental filtering, etc. While these services should be provided by the endpoints, it is not always optimal or possible due to the service requiring network visibility or because the endpoint does not have enough resources. Thus, the multi-context TLS presented in this work aims to securely and explicitly include these middleboxes in TLS sessions instead. Moreover, it allows entities to dynamically choose which portions of content are exposed to the middleboxes (e.g., content headers vs. body), to allow select middleboxes to modify the data while retaining its authenticity and integrity through read and write permissions, and to be deployed incrementally.

While this work does not deal directly with user authentication, it shows how important the role of middleboxes can be, especially when integrated carefully. This work adds weight to the approach used in this thesis, where a connection to access data in a database is enriched with additional services using services that stand between the user and the database.

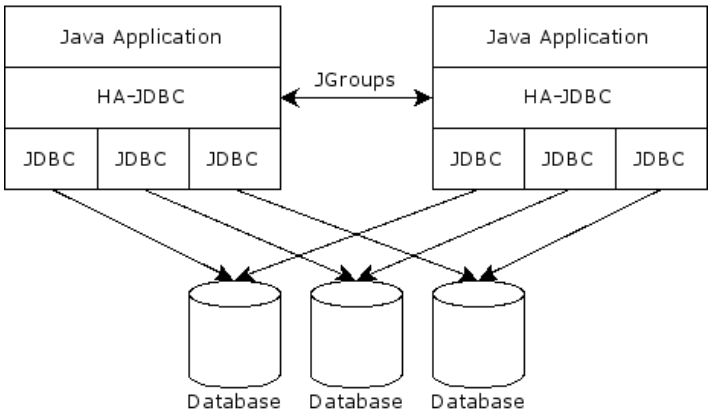


FIGURE 3.4: HA-JDBC overview [103].

3.2.6 High-Availability JDBC

High-Availability JDBC, or HA-JDBC, [103] is an existing JDBC proxy project that has been implemented, is readily available and provides many features on top of what JDBC currently supports. Fig. 3.4 illustrates how it works.

HA-JDBC functions as a middle-layer between the Java application and the JDBC objects that connect to the databases. In doing so, it can manage a connection to each database in the cluster while providing the application with an interface that is identical to JDBC. When clusters are accessed by multiple Java applications, cluster membership changes are distributed using a JGroups channel.

However, HA-JDBC only focuses on being light-weight, transparent, and providing fault-tolerant clustering capabilities to the underlying JDBC driver. Therefore, it still requires the client application to use the database credentials. In this light, it differs from the work done for this thesis in which the client authenticates with the proxy itself using a server-generated token and never uses the database credentials directly.

3.2.7 Biometrics in Authentication

User biometric data could be used to prevent client applications from having to store the database credentials. However, it is not necessarily safer or more convenient than the alternatives [104].

For example, if a system uses face recognition to authorize users, then a malicious user could attempt to find a photo of a legitimate user online and use that to bypass the system. The system could be tweaked to detect when a photo is being used and deny access in those scenarios, but that increases the likelihood of false negatives. Consequently, some legitimate users might be denied access and the usefulness of the system reduced.

Other kinds of biometric data have their problems. Databases containing this data can be leaked [105], which similarly to password database leaks can allow malicious users to access the exposed systems. This thesis focuses on protecting database credentials in the form of passwords from this kind of exposure, but it could be potentially adapted to protect biometric data if necessary.

3.3 Summary

In this chapter, various works related to access control models and various other facets of security were presented and discussed. These included traditional access control models, which are related to the base architecture used in the previous work and whose capabilities were found to be lacking to ease the development of data access applications. Included as well were access control models based on soft requirements, which were shown to be very specific in scope.

Confidentiality concerns were also discussed, given the possibility that disclosing database schemas could provide malicious users with the information they need to perform attacks with a higher chance of success or to infer sensitive information from carefully crafted queries. Authentication and communication standards and works were also discussed as shown to either be ineffective, difficult to set up or not adequate for some scenarios.

All of these issues are present in this thesis as well in search of the goal of expanding the application scenarios for access control, however, they were addressed in ways that always complement or enhance the solutions found in the current state of the art. The remote database access methods found lack a way to indicate to a developer when access to a method has been lost until an error occurs at runtime, which this thesis aims to resolve with policy-aware data access interfaces. Furthermore, every single soft access control system was a specialized system for a particular scenario and lacked a proper way to validate that the policies written for them were correct. This thesis aims to complement these works by presenting a soft access control model with general applicability and to provide a mechanism to validate its policies. Finally, no real methods to protect database credentials in database applications was found, which are usually just stored with them and could be stolen if the applications are exploited. This thesis aims to enhance this aspect by designing a new authentication scheme for databases integrated with the remote execution.

Chapter 4

Secure Remote Data Access

There are many layers to accessing remote data, and many of them suffer from issues that not only increases the development time of applications that need to access data remotely, but also introduces potential security vulnerabilities if the developers are not thorough, such as how to store the data store credentials. This chapter presents and discusses the research and advances made towards easing the development burden of application that access remote data.

Since vulnerabilities are going to be identified and addressed, it is important to know who are the expected attackers and how they are going to exploit a remote data access system. The client in this model is a database application that is exposed to public medium, like a web-server connected to the internet. The server, then, is the database server itself. With this scenario in mind, it is clear that the client host machine could be exploited if improper configuration or out-of-date libraries introduces a vulnerability into the system. Here, an attacker on the internet could connect to the client, exploit the vulnerability and potentially access the client database application. However, another scenario is also considered. A rogue employee or a business where an attacker can infiltrate and access the internal network could access the client database application and even the database server. This way, they could eavesdrop the network or perform man-in-the-middle attacks to learn database credentials to connect to the database directly.

The general requirements surrounding the Secure, Dynamic and Distributed Soft Access Control Architecture (S-DiSACA) iteration have emerged from the research done to ease the development burden of relational database applications so that developers do not have to master the database schema to write database operations or about securing their client applications for the scenarios described above. This is the major benefit that this architecture brings over other solutions that a developer may use.

To illustrate the point of the burden of having to the learn the database schema, consider the Structured Query Language (SQL) query shown in Listing 4.1.

LISTING 4.1: Example SQL query.

```
SELECT Address
FROM Customers
WHERE CustomerName = ?
```

This very simple query requires the developer to know not only the name of the table where the customer data is stored (Customers) but what data is stored there and the name of the corresponding columns (CustomerName and Address). If the context is extended to include tens or hundreds of tables, this quickly becomes a problem.

Notice as well how the query accepts one parameter, the customer name. This filters the output of the query so the address returned matches that of the customer name. However, database connectivity tools do not apply any kind of restrictions over the values passed to this parameter. Application logic can be used to regulate which values are used, for example by allowing a user to select values obtained from a previous query. Regardless, if the application is vulnerable someone could be able to use reflection mechanisms[106], [107] to inject their own values into the parameter. This would allow them to access data that otherwise should not be available.

Furthermore, behind this query are access control processes to verify that the application running this query is authorized to access the data, which can generate runtime exceptions if it is not. These exceptions only occur at runtime, so they are impossible to detect normally during compilation. Considering that queries can be much more complex, the problem can only be compounded further.

Another issue arises when considering that to run this query, the application must have an open connection to the database and that it must be authenticated. This is usually achieved using standard database connectivity tools such as Java Database Connectivity (JDBC), and the authentication is performed via a username and password credential pair. However, these credentials must be stored somewhere in the application, the host that runs it, or provided by the user of the application. In the first two cases, the credentials are prone to be stolen if the host machine is breached or if it is running in a semi-public location where anyone could access it, such as a reception desk. In the latter case, different issues apply. The user may select a weak password or if forced to use a strong password, write it down somewhere so it cannot be forgotten. Password policies have been designed to try to counter this issue [108]. Unfortunately, they are rarely applied.

Another point of contention is that the database connectivity tools used rarely support any kind of encryption. Internal attacks on the networks could expose the data being accessed via eavesdropping or even modify it via impersonation attacks.

Thus, the general requirement for the S-DiSACA is to address all of these issues by providing the following solutions:

- Operation Execution Protection: operations and parameters should be protected so that only allowed values can be used and operations cannot be modified to produce different results;
- Operation Sequencing: following the previous solution, some operations are meant to provide values for another. Therefore there should be a way to regulate the order in which operations are executed to prevent unexpected outcomes;
- Data Store Credential Protection: a solution to prevent the database credentials from being stolen by exploiting any one single component in the S-DiSACA;
- Communication Security: a secure method to transport data between the client application and the database should be made available.

Each of these points are discussed in this chapter, which is divided as follows: section 4.1 introduces the steps taken to protect both the database operations and the parameters passed in those operations; section 4.2 shows how operations can be sequenced to have use cases explicit during development; section 4.3 details the enhancements made to protect the database credentials and the architecture security as a whole; finally, section 4.4 provides a summary of the research performed.

4.1 Operation Execution Protection

In this section, the method by which the developer burden or protecting the operations and parameters from modification with the S-DiSACA is detailed. This topic is tackled in two fronts: first, how to protect the operations; and second, how to protect the parameters used by those operations.

Protection in this context refers to making the operations immutable from those that are defined in the access control policies, and to assure that the values passed to these operations are also immutable when their origin is another operation.

To illustrate this point, consider the code shown in Listing 4.2 that shows some example Java code that accesses a database to retrieve a list of patients, from which a doctor can select one to see its history.

LISTING 4.2: Example code for a medical database scenario.

```
1  PreparedStatement selPatients = conn.prepareStatement(
2      "SELECT id, Name FROM Patients WHERE DoctorToken = ?"
3  );
4  selPatients.setString(1, token);
5  ResultSet patients = selPatients.executeQuery();
6
7  int patientId = displayAndSelectPatient(patients);
8
9  PreparedStatement selHistory = conn.prepareStatement(
10     "SELECT * FROM PatientHistory WHERE PatientId = ?"
11 );
12 selHistory.setInt(1, patientId);
13 ResultSet patientHistory = selHistory.executeQuery();
14 displayPatientHistory(patientHistory);
```

The first three lines prepare a SQL select query to be executed. In line 4 the application sets a doctor identifier (e.g. a token obtained from a smart card that uniquely identifies the doctor) and in line 5 the query is executed and the results saved in the variable *patients*. The doctor then selects one patient, and the application retrieves its associated id. Then the application prepares the next select query to obtain the patient history in lines 9 to 11 and sets the selected patient id in line 12. Finally, it executes the query and saves the history of the patient in the variable *patientHistory* in line 13 and displays the information in line 14.

This example shows several potential problems. On the one hand, the parameters are not restricted in any way, allowing a malicious user to modify the program to potentially alter the parameter values.

```
10  public interface IWikiPagesAPI {
11      public APIResult updateArticle(APIResult article, String updArticle);
12      public APIResult getArticleByTitle(APIResult articleTitle);
13      public APIResult searchArticles(String query);
14      public APIResult getNumArticles(String query);
15      public APIResult getLastModifiedArticles(Integer limit);
16      public APIResult getArticleModificationCount(APIResult article);
17      public APIResult deleteArticle(APIResult article);
18      public APIResult getArticlePreviousVersions(APIResult article);
19      public APIResult revertArticleChanges(APIResult article, APIResult prevVersionArticle);
20      public APIResult getUsers();
21      public APIResult banUser(APIResult user, Long timeInMillis);
22      public APIResult unbanUser(APIResult user);
23  }
```

FIGURE 4.1: Example Database Access API.

On the other hand, and perhaps worse still, the queries are hard-coded into the application logic. This means that the database accepts any query sent to it for execution, so the application could be modified maliciously to execute different queries. Each of these problems are discussed in turn.

4.1.1 Operation Protection

Considering the hard-coded queries problem, one simple idea to avoid them is to use stored procedures, which are subroutines available on the database that execute one or more queries. This way, the application needs only to know the stored procedure names to execute the associated queries and since the actual queries are stored in the stored procedures that cannot be modified from the application side, they are protected.

This was the approach used in the Secure, Dynamic and Distributed Role-based Access Control Architecture (S-DRACA), which was enough when dealing with only relational databases. However, the first research question (RQ1) mentioned in section 1.3 was intended to break this dependency with relational databases, so the S-DiSACA has a wider range of applications. Since it is not known what type of underlying data storage used, stored procedures may not be available. Thus, the stored procedure solution needs to be adapted.

The simplest solution is to implement a feature similar to stored procedures, where the client applications receive function labels that they can execute. These functions are then implemented in a data store application programming interface (API) that resides in the Policy Manager, the S-DiSACA server-side component. This is an approach very similar to remote method invocation (RMI) [109], except it is enriched with several security features that are not usually considered in RMI, namely the parameter protection, communication security, and the operation sequencing that are discussed later in this chapter.

The first step to implement an RMI-based solution is to implement a database access API to be made available to the client applications. These APIs are specific to each application scenario and implement the data access logic to the particular data store in use. Fig. 4.1 shows one such API interface.

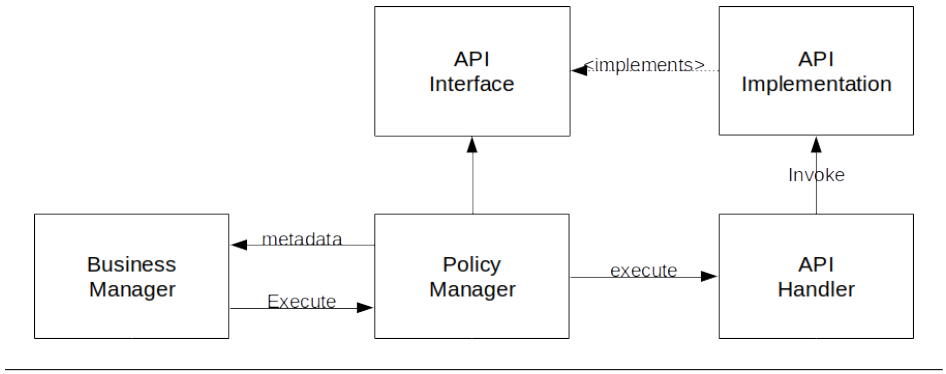


FIGURE 4.2: Operation protection block diagram.

This API allows to access and manipulate a Wikipedia dataset of articles and users. As mentioned, the implementation of this API is specific to the application scenario, and thus it is not very relevant. Suffice it to say that the code executes the database-specific procedures to achieve the intended outcome of the API method and instantiates an APIResult object. An APIResult object is a generic object that handles API results and augments them with security features that are discussed in a later section.

Fig. 4.2 shows a block diagram of the operation protection solution. Once the client application Business Manager connects to the Policy Manager for compilation, the Policy Manager sends the public methods in the registered API as metadata. The Business Manager on the application side then recreates the API interface, allowing the client application to use it. Its implementation is quite different, however, as it simply sends an "Execute" command to the Policy Manager, along with the method name and potential parameter values. Upon reception, the Policy Manager uses an API Handler that executes the intended method on the registered API via Java Reflection and returns the generated APIResult object after several security checks.

This approach also has some implications in the S-DiSACA lifecycle, as the implemented API to access the database now needs to be processed to extract the public methods to be sent to the client application as metadata. The client application then also uses a different interface and implementation strategy. These changes are further detailed in section 6.1 where the overall final architecture is presented.

4.1.2 Parameter Protection

The other issue left to consider is the potential modification of the parameters passed to the queries, which can differ depending on their origin. Parameters can be user-provided, in which case they cannot be protected, or they can come from data obtained in previously executed queries.

Fig. 4.3 illustrates how the parameter protection was achieved in the previous iteration. It shows an initial business schema (*S_Cust_1*) being instantiated (line 75) and executed (line 76) to obtain a list of customers. Then, a second business schema (*S_Orders*) was also instantiated (line 77) and then also executed (line 79) to obtain a list of orders placed by a customer and shipped to a specific country.

```
75      S_Cust_1 = factory.get_S_Customers_all(session);
76      S_Cust_1.execute();
77      S_Orders = S_Cust_1.nextBE_S1(S_Orders_byShipCountry, session);
78      // Use data from S_Cust as input
79      S_Orders.execute(S_Cust_1, "Portugal");
```

FIGURE 4.3: S-DRACA parameter protection example code.

Note that the first business schema is passed to the execution of the second (line 79), along with a second parameter ("Portugal"). In this scenario, the *S_Orders* business schema retrieves the current value in *S_Cust_1*, and since it was not modified it is the first customer returned by the database.

This example also showcases another aspect of this problem. When the values for some parameter are not obtained through a previous business schema, the client application has no choice but to either pass it hard-coded or to request the value to the user (e.g. "Portugal" in line 79). There is nothing that can be done in these cases. Thus, sensitive parameters should have their values obtained from the database to restrict the range of possible values while ensuring that any parameters requested to the user cannot be used to disclose any sensitive information.

However, this approach does not protect against malicious intent, as reflection mechanisms can still be used to modify the values stored in the *S_Cust_1* business schema before passing it to *S_Orders*. Preventing the queries from being executed with invalid parameters is one of the goals of the S-DiSACA, and as such two modifications to this approach are presented:

1. Build an internal cache of business schema results;
2. Have both the Business Manager and the Policy Manager manage their own cache, so both can check for input value modifications.

With a cache, the system is capable of automatically obtaining the parameter values that rely on previous operation executions. Thus, by keeping the same record of the returned results on the server-side Policy Manager, it can verify if the parameter value received is contained in one of the previous records. If it is not, then the execution of the operation is stopped and the potential malicious intent thwarted. A client-side cache is also used so that any legitimate client application cannot send an invalid parameter value to the server, reducing server load while easing the development process. While the client-side cache remains susceptible to reflection attacks, the server-side cache is not accessible from the client applications, increasing the trust on the validity of the parameters.

It is worth noting that the cache is not required to maintain every result sent to a client application forever. Since operations are meant to be executed in the context of a sequence and the data retrieved is not meant to be used outside that context, once a sequence is completed the cache for that client application can be emptied. This operation sequencing feature is introduced in the next section to explain how it was achieved and how data flows between operations.

4.2 Operation Sequencing

The issues addressed in this chapter until now have dealt with protecting the operations and their parameters so that if they are maliciously modified they cannot be executed on the data. However, a more subtle approach that can lead to sensitive information being disclosed can still be carried out that most developers are not even aware of. If the system defines the operations that can be executed, the order in which they are executed is normally not considered. This can be exploited to leak sensitive information, as shown in [110].

Thus, to avoid the operations allowed to the user to be used in unexpected contexts, an initial iteration of operation sequencing was implemented in the S-DRACA, where each sequence was defined as a particular use case in the application scenario. This iteration, however, only supports sequences of operations that did not have any divergence, i.e. two or more operations could not be allowed to be executed after another at the same time, only one. Not only that, these sequences were dependent on the Role-Based Access Control (RBAC) policies, which are also intended to be generalized.

Thus, the concept of operation sequencing is generalized and modelled using graph theory. These ideas have also been used to design a sequence-based access control model, described in [31]. Thus, a sequence in this context is defined as follows:

- By a set of actions A and their input parameters P;
- By a set E of directed transition relations between actions;
- And by a set of users U allowed to execute the sequence.

The set of users can be explicitly defined, or implicitly through some condition that they must satisfy, such as playing some role, possess the correct set of attributes, etc. This way, a policy for the S-DiSACA only needs to associate permissions with a set of sequences, and since permissions are granted or denied to each user automatically it is possible to determine which sequences they are allowed to execute.

Consider the following use case in an online shop. First, a client is authenticated into his online account (A), and then checks his existing cart of items (B). However, his method of payment needs to be updated, and so he proceeds to update it (C). Finally, the client can proceed to checkout and order the items on the cart (D). This scenario can be defined as a sequence of actions, as shown in Fig. 4.4.

The client started with action A, authenticating itself, which is mandatory. Then he opened the cart to verify the items within and to proceed to checkout, which is also considered mandatory. This is shown in the figure as a directed relation from A to B. Then he noticed that his payment option needed to be updated, a step that is not always necessary before checkout, but one that can happen. Thus, two directed relations exit from B, one to C where the client updated their payment option, and one to D to finalize the checkout. Finally, to allow the client to checkout right after updating the payment option, a directed relation also exists from C to D.

If a malicious user can breach the client application and attempts to execute the operation associated with action C to access the payment options in the database, he is unable to do so as action C is in the

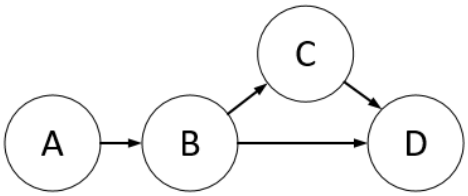


FIGURE 4.4: Online shop sequence of actions example[31].

middle of a sequence and the Policy Manager should report a sequence violation. If the malicious user tries to follow the sequence instead, action A requires him to authenticate, which means that he must have stolen the authentication credentials as well. Finally, even if he can follow the sequence, action C can be defined to automatically use the identifier of the user returned by action A. If the malicious user attempts to modify it, the Policy Manager is able to detect the change through the parameter protection feature discussed previously and deny the execution. Modifying any of the operations associated with the actions is not possible either due to the operation protection feature.

4.2.1 Definitions

The operation sequencing was formally defined in [31], where it was used to propose a sequence-based access control model. The same ideas were applied in the Sequence Manager in the S-DiSACA, and thus the concepts and definitions behind the graph-based approach are explained in this section.

Formally, at the core of this approach is a set of action flowcharts that are mapped to the users that are allowed to use them. First, consider Eq. 4.1 and Eq. 4.2 that define a set of actions and a parameter, respectively.

$$A = \{a_1, a_2, ...a_N\} \tag{4.1}$$

$$P = (name, datatype) \tag{4.2}$$

Each action is associated with an operation that can be executed, and a parameter is a tuple that contains the name associated with the parameter and its datatype. With this, the set of action nodes V in an flowchart is defined as shown in Eq. 4.3:

$$V = \{(a, \{P\})\}, a \in A \tag{4.3}$$

A node in an action flowchart is then comprised of an action and a set of parameters that it accepts, and it allows a user authorized to execute it to access or modify the data.

However, the actions alone are not sufficient to define the flowcharts. A set of directed relations between the actions is still needed. These directed relations can be defined using ordered 2-element tuples from V, and a set E of these tuples defines the directed relations between action nodes. Eq. 4.4 defines E.

$$E = \{(u, v) : u, v \in V\} \tag{4.4}$$

Eq. 4.5 defines a flowchart G given the previous definitions, which is a tuple of the set of action nodes V along with the set of directed relations E between them.

$$G = (V, E) \tag{4.5}$$

The additional functions $TransitionSet(G)$ and $ActionSet(G)$ are used to refer to the set E and the set V from a flowchart G, respectively.

In any given application scenario, each of these flowcharts G are used to define a distinct use case. Thus, a set of flowcharts SOF is defined in Eq. 4.6 which contains every flowchart G. Finally, Eq. 4.7 defines SOF_u, a subset of SOF that a user u in the set of authorized users U is given access to. SOF_u determines the entire set of actions that the user U is allowed to execute and in which order they can be executed.

$$SOF = \{G_1, G_2, ...G_M\} \tag{4.6}$$

$$SOF_u \subseteq SOF, u \in U \tag{4.7}$$

However, a way to track a user along each sequence is also needed so the system can validate if the execution of a particular action node is allowed or not. Moreover, the method by which this tracking is updated between action node executions is also important to define, as the set of available parameter values is important for parameter protection purposes. Thus, Def. 4.1 defines a User Access Pointer (UAP) which is used to determine in which flowchart and action node a user is currently on.

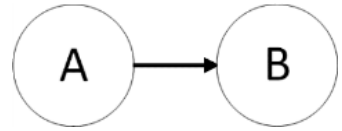
Definition 4.1. *User Access Pointer:* Given a SOF, the UAP is a tuple of elements (G, v) that uniquely identifies a flowchart $G \in SOF$ and the current action node $v \in V$ the user has used.

A user that has not begun executing any sequence of operations is said to have a UAP on step 0 (UAP_0) and it does not reference any flowchart or action node. When the user selects a flowchart to execute, the UAP is updated to step 1 (UAP_1) and it references the root node of the selected flowchart. The process of updating the UAP to allow a user to execute different actions is known as *Stepping* and it must always follow a directed transition in the flowchart, as defined in Def. 4.2:

Definition 4.2. *Stepping:* Consider a flowchart G and that its UAP is on step n of the flowchart traversal, denoted UAP_n. Stepping is a process in which UAP_{n+1} is generated by satisfying the implication in Eq. 4.8:

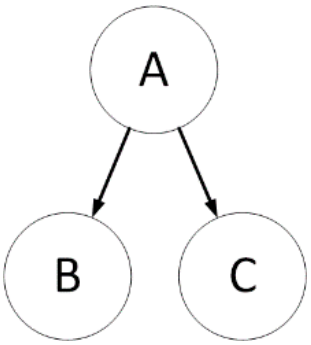
$$\forall_x \forall_y (UAP_n = (G, x) \wedge UAP_{n+1} = (G, y) \Rightarrow (x, y) \in TransitionSet(G)) \tag{4.8}$$

There are many scenarios in which *Stepping* can occur depending on the in-degree and the out-degree of the action nodes, i.e. the number of relations going into and out of a node, respectively.




```
1 execute A
2 execute B
```

FIGURE 4.5: Trivial stepping example with pseudo-code[31].




```
1 execute A
2 if(condition) then
3    execute B
4 else
5    execute C
6 end if
```

FIGURE 4.6: Splitting stepping example with pseudo-code[31].

These scenarios also correspond to particular use case logic scenarios, and the relation between both are explored in the next section.

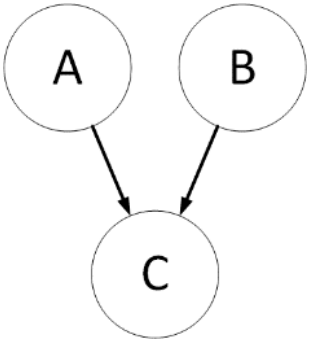
4.2.2 Stepping Scenarios

This section describes the various scenarios in which the *Stepping* procedure can occur and translate them to the equivalent pseudo-code.

Fig. 4.5 shows the simplest type of *Stepping* that can appear, and it is the trivial case in which the out-degree of node A and the in-degree of node B are both equal to 1. If UAP points to node A, then by Def. 4.2 the user is forced to execute the action in node B next since the relation (A, B) is the only relation that can satisfy it. The pseudo-code shows that this case is equivalent to executing the actions in the nodes sequentially.

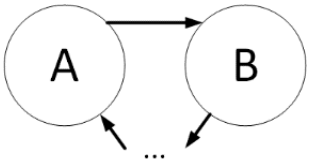
Fig. 4.6 shows the second possible case that can appear, called *Splitting*, where a node has an out-degree of 2 or higher. This means that the execution branch can split, because if the UAP points to one such node then there is more than one relation that satisfies Def. 4.2. If the UAP points to node A in Fig. 4.6, then the user can execute either the action in node B or C. This translates into pseudo-code as a conditional branch in the application logic. After executing A (line 1), then the application either executes B (line 3) or C (line 5) based on some condition (line 2).

The next case is the reverse of the previous case, where a node has an in-degree of 2 or higher. Fig. 4.7 shows this scenario, called *Merging*. This means that the execution branch can rejoin or synchronize, because the UAP has to eventually point to that node, regardless of the node that precedes it. To illustrate, if the UAP points to node A in Fig. 4.7, then the user has execute the action in node C next.




```
1 if(condition) then
2     execute A
3 else
4     execute B
5 end if
6 execute C
```

FIGURE 4.7: Merging stepping example with pseudo-code[31].




```
1 while(condition) then
2     execute A
3     execute B
4     ...
5 end while
```

FIGURE 4.8: Cycling stepping example with pseudo-code[31].

If UAP pointed to node *B* instead, the same would be true. This translates into pseudo-code as a conditional branch in the application logic that leads to further logic that is common to both branches. After executing *A* (line 2) or *B* (line 4) based on some condition (line 1), the application would always execute node *C* (line 6).

Finally, transitions do not need to point always towards new nodes. When a node has a transition to another node that has already been executed, then the flowchart contains a *Cycle*. Fig. 4.8 shows this scenario, where any number of nodes can exist in the cycle. This is not a new *Stepping* case, but it translates to a new logic construct in the pseudo-code, where the same actions (lines 2-4) are executed while the condition remains true (line 1). This also allows for the same node to be executed any number of times sequentially if a node has a relation to itself, creating a *Loop*.

All these scenarios can be mixed to create complex use cases that can easily be translated into the application logic. The benefit of this approach is that this logic is stored in the access control policies and not in the application itself, which prevents malicious users from being able to bypass it. However, given that the parameter protection feature requires data from each node execution to be available to nodes later in a flowchart, it is important to define how this can be achieved.

4.2.3 Information Flow

This section discusses the information flow within a flowchart and between flowcharts, which allows for values obtained from past accessed nodes to be used by other nodes. This process is essential to support the parameter protection feature as discussed in section 4.1.

Since the information that is allowed to flow must come from nodes that have been accessed, the definition of an accessed node is given by Def. 4.3:

Definition 4.3. *Accessed*(G, v): A node $v \in ActionSet(G)$ for a given flowchart G is said to have been accessed when a user's UAP possessed a reference to node v on at least one step leading up to the current step N as shown in Eq. 4.9:

$$\forall_{v \in ActionSet(G)} \exists_{n \leq N} (Accessed(G, v) \Rightarrow UAP_n = (G, v)) \tag{4.9}$$

Additionally, the information that flows within a flowchart also depends on the user that is executing it, since a different user may execute different nodes depending if there are any *Splitting* scenarios. Thus, a User Context is also defined in Def. 4.4:

Definition 4.4. *User Context:* Given a user u and each graph G from its set of flowcharts SOF_u, its user context (UC_u) is a pair of elements containing its current UAP on step N (UAP_N) and the set of nodes previously accessed by user u as shown in Eq. 4.10:

$$UC_u = (UAP_N, \{v : v \in (ActionSet(G) \wedge Accessed(G, v))\}) \tag{4.10}$$

The set of nodes accessed by a particular user is referenced using the predicate $AccessedSet(UC_u)$. Thus, the UC_u is updated every time a *Stepping* occurs and it can be used to obtain the values to parameterize the data access. An action is only required to specify which parameters receive their value from another node (in the parameter type) and from which node (in the parameter name). This way, the process of passing parameter values from other nodes can be automatic, ensuring that the user cannot provide invalid or unexpected values either by mistake or with malicious intent.

When a user completes the execution of a flowchart, the UC_u needs to be reset back to step 0 to allow for another flowchart to be executed and to prevent data to be used outside of its context (i.e. in another flowchart). This process is shown in Eq. 4.11:

$$Reset(UC_u) : UC_u = (UAP_0, \emptyset) \tag{4.11}$$

Finally, since the flowcharts are meant to define sequences of operations that address real-world use cases, the idea of reusing flowcharts that resolve/perform a common issue/task in more complex flowcharts naturally follows.

Inter-Flowchart Stepping

Fig. 4.9 shows an example of an inter-flowchart scenario, in which the node labelled A' is a reference to the initial node of another flowchart.

In this situation, when *Stepping* from one flowchart to another, the current UC_u is saved in a stack and a new UC'_u created for the sub-flowchart. The sub-flowchart then uses the new UC'_u as normal until it terminates, in which case any data required by the parent flowchart should be returned to it. This process is shown in Eq. 4.12 where the original UC_u is updated to contain the data it contained and the data returned by the sub-flowchart in its UC'_u.

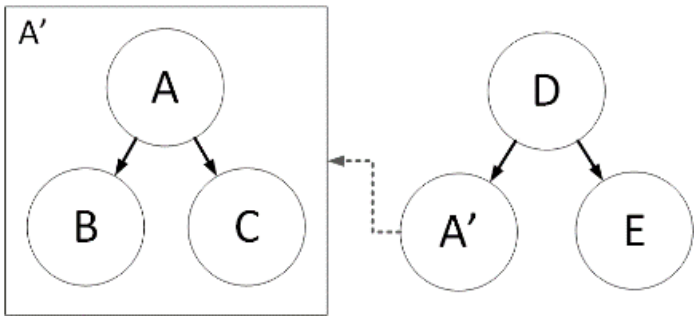


FIGURE 4.9: Inter-flowchart example[31].

$$UC_u = (UAP_n, AccessedSet(UC_u) \vee AccessedSet(UC'_u)) \tag{4.12}$$

Note that the UAP of the original UC_u, which was on step n, remains the same throughout the execution of the sub-flowchart. This UAP_n allows the system to know to which flowchart and which node within that flowchart it should return to after the sub-flowchart is executed and terminates.

Information Flow Restriction

Since some flowcharts can be used as part of more complex flowcharts, the information flow within those more complex flowcharts should be controlled when executing passes to the sub-flowchart.

This control is provided by revocation lists, as defined in Def. 4.5.

Definition 4.5. *Revocation List:* Given a flowchart $G \in SOF$, a revocation list R is a set of previously accessed nodes in $ActionSet(G)$ that must prevent further access to their data, as shown in Eq. 4.13:

$$R = \{v : v \in ActionSet(G) \wedge Accessed(G, v)\} \tag{4.13}$$

Given this new set of nodes whose results can no longer be accessed, the UC_u definition should be updated to exclude the nodes in the revocation list R, as shown in Eq. 4.14:

$$UC_u = (UAP_N, \{v : v \in (ActionSet(G) \wedge Accessed(G, v)) \setminus R\}) \tag{4.14}$$

The revocation list R can be stored in either the transition relations between nodes (Eq. 4.15) or in the nodes themselves (Eq. 4.16). However, for inter-flowchart purposes storing it in the transition relation is preferable, as the revocation list can be personalized to each flowchart that calls another more easily. This also applies to when a sub-flowchart terminates, allowing to specify exactly what data should be made available to the parent flowchart as it transitions back.

$$E = \{(u, v, R) : u, v \in V\} \tag{4.15}$$

$$V = \{(a, \{P\}, R)\}, a \in A\} \tag{4.16}$$

Furthermore, two types of inter-flowchart *Stepping* are defined: dependent and independent. The dependent type requires information from the parent flowchart to execute, and thus require the list of accessed nodes from the original UC_u to be copied over to the sub-flowchart user context UC'_u. Note that the approach mentioned above to revoke any unnecessary information when copying the UC_u can be used. However, this type of inter-flowchart *Stepping* is discouraged, as it prevents the sub-flowchart from being used on its own. The independent type, as the name implies, does not require any data from its parent flowchart, and so it can use a freshly created UC_u with an empty set of accessed nodes.

4.3 Architecture Security

Having discussed how the remote execution of operations can be enhanced from its previous iteration, the question now turns towards the authentication and authorization of the client applications, an aspect of application development that tends to be left to the last minute but that can be disastrous if done incorrectly.

In previous iterations, the client applications would simply connect directly to the database. This had several security issues, namely that the database credentials were hard-coded into the client applications, the database had to be directly accessible, and the communication between every component in the system was not encrypted, allowing several attacks to be possible (e.g. man-in-the-middle attacks, impersonation, network eavesdropping, etc.).

As discussed in section 3.2, solutions such as Transport Layer Security (TLS) and Virtual Private Network (VPN) can be used to authenticate and authorize applications. However, they are not without their drawbacks. In the case of TLS, authentication is achieved using digital certificates and the PKI. However, CA issued certificates are not free and trust must be placed on the CA that it won't issue the same certificate to other entities. In the case of a VPN, the VPN server only protects the internal networks from unauthorized access from external networks. Thus, attacks that occur on the internal networks are not within this scope, and the VPN server is a single point of failure security-wise.

Thus, the last iteration introduced the concept of pushing the credentials to access the database to the server-side while allowing client applications to still use JDBC to access the database. This solution, however, goes against the research question RQ1 as the system was not easily adaptable to new interfaces and relied heavily on JDBC. Furthermore, the credentials could still be obtained if the authentication server was compromised. A solution based on applying a pre-shared key to an established TLS connection was also proposed to counter the PKI related issues discussed. Unfortunately, this solution relies on reflection mechanisms to accomplish such a task, as TLS connections based on pre-shared keys are not usually available, and the solution breaks for newer versions of Java. Thus, modelling how the pre-shared key is applied to a TLS connection is important to allow future implementations of TLS to potentially include this approach.

The solutions developed for the S-DiSACA to address the issues discussed and achieve the following tasks are introduced in this section:

1. The credentials must remain protected even if one component in the architecture is exploited;
2. Connections to the Policy Manager can only be made after a client successfully authenticates with an authentication server;
3. The mechanism that calculates new connection keys using a pre-shared key should be formally defined, and a fallback solution proposed in case the application of the pre-shared key fails via reflection.

4.3.1 Credential Protection

A method to protect credentials such that a single component being exploited cannot leak them to a malicious user is discussed in this section, along with how connections to the data store via the S-DiSACA can only be attempted after a user has been successfully authenticated.

Since the idea is to create an authentication scheme where the database credentials are protected from being disclosed from the application, it is clear that these credentials need to be stored elsewhere. Thus, the initial approach was to create an authentication service that stores them and provides them to the data access server once the application is authenticated. The work presented in [35] and [36] explains this procedure, bridging the previous JDBC-based iteration to the security requirements discussed above. This procedure is explained here taking into consideration the research question RQ1 while addressing RQ2 and RQ3, meaning the JDBC proxying is replaced by a generic data access API. Since this API is generated on the client application much like the JDBC API was, this does not influence the approach described and the results achieved in the published works.

Fig. 4.10 shows a diagram of the credential protection generic architecture for the S-DiSACA, and it is composed of three major components:

- The **Policy Manager**, which connects to the data store and handles API method execution requests from the client application;
- The **authentication service**, which authenticates and generates a token for each client application. It can also request the Policy Manager to open an endpoint to which the client application may connect and present its token;
- The **client application**, which contains no relevant information for authentication with the database. However, the user of the client application must know the credentials to the authentication service.

Thus, the presented solution must be able to protect the database credentials even when either the Policy Manager or the authentication service are compromised. The only exception is if the data store itself is compromised, in which case access can be obtained without proper authentication.

To achieve this goal, both the Policy Manager and the authentication must be given an asymmetric key pair each, and the public component of those keys disseminated between them. Furthermore,

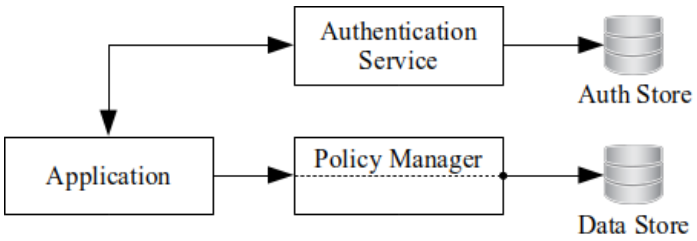


FIGURE 4.10: Credential protection architecture diagram.

each client application is also provided with a unique symmetric key C when it is registered in the system. These keys and sensitive information is divided as follows:

- The **Policy Manager** has access to the database credentials, encrypted with the client symmetric key C, and its own private key.
- The **authentication service** has access to the symmetric key C, encrypted with the Policy Manager public key P_{pub} via RSA, and the client credentials (username, MD5(password + salt), salt, $P_{pub}(C)$).

Having defined the overall architecture for credential protection, the communication protocol and an attack scenario analysis follow.

Connection Protocol and Key Usage

The connection and authentication procedure that a client application must follow with the architecture is shown in Fig. 4.11. Note that every request is acknowledged and any errors encountered terminate the process.

The client application initiates the connection process by connecting to the authentication service and establishing an encrypted communication channel. This process is described in the next section.

The client application then requests the salt value associated with its password, so that it can generate the password hash $MD5(p + salt)$ to authenticate with the authentication service. If the authentication succeeds, the client application can then request a token T that allows it to connect to the Policy Manager. The authentication service, upon receiving this request, asks the Policy Manager to open an endpoint for the client. The username is sent both encrypted and signed, so the Policy Manager can guarantee that the authentication service sent the request and that only it can read the data. Once an endpoint is open, the associated port p is sent back to the authentication service in a similar manner.

Finally, the authentication service can generate the token T and send it to the client application. The client application then connects to the Policy Manager, establishing an encrypted communication channel. The client application then presents the token T that identifies it, and after the Policy Manager carefully validates its contents, the process of generating the operation flowcharts and everything else can follow.

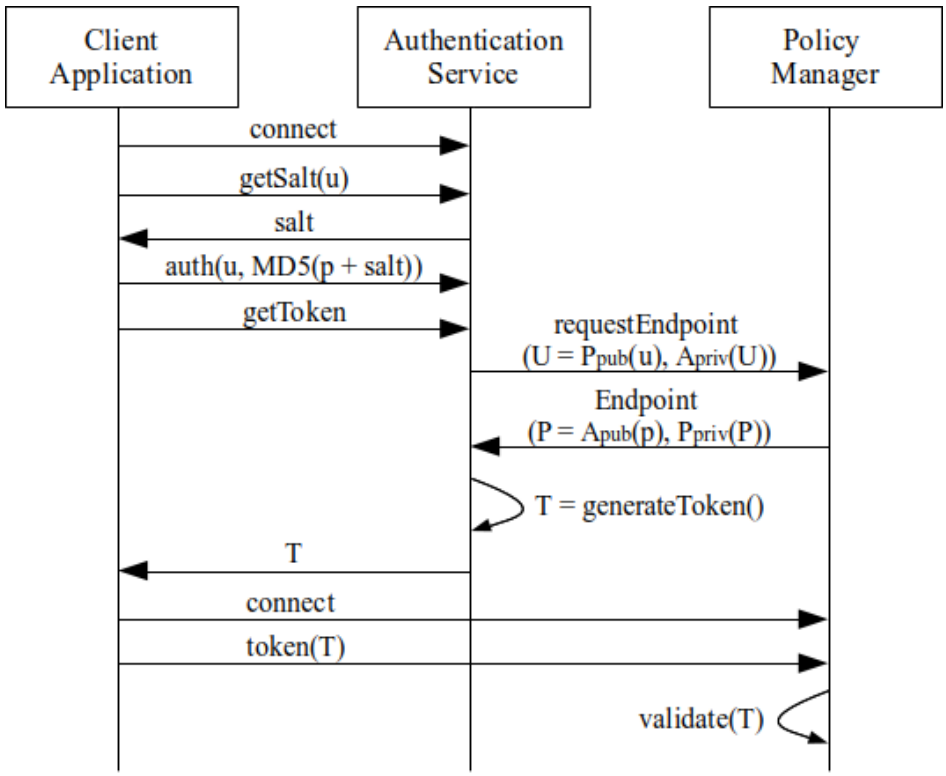


FIGURE 4.11: Connection protocol diagram using credential protection.

The token T generated by the authentication service is used to identify the client application when it attempts to connect to the Policy Manager. This token is necessary because the Policy Manager requires more than just a pair of identification credentials from the client application. As mentioned in the previous section, it requires the symmetric key C stored in the authentication service to decrypt the database credentials associated with the client application. Thus, the token T is generated with the following fields:

- **username:** The name of the client application attempting to connect. This is a field encrypted with the Policy Manager public key P_{pub}.
- **created:** The timestamp when the token was created.
- **expires:** A timestamp of the instant when the token expires and should no longer be accepted.
- **nounce:** A random 32-bit integer.
- **endpoint:** The IP address and port to which the client application was authorized to connect of the Policy Manager.

- **C:** The symmetric key associated with the client application, required to decrypt the database credentials stored in the Policy Manager. This is a field encrypted with the Policy Manager public key P_{pub}.
- **sig:** The authentication service signature of the token. Obtained by hashing the token and encrypting the result with the authentication service private key A_{priv}.

The *username* and *C* fields are encrypted so that only the Policy Manager can read them, while the *sig* field allows the Policy Manager to validate that no field in the token was modified in transit.

Server Attack Analysis

An analysis of this approach considering several attack scenarios is provided in this section to demonstrate that the discussed requirements are satisfied.

To do so, let us consider which sensitive information is disclosed when the data stored in each of the servers and the client application are disclosed in turn. As a reminder, it is only expected that the database credentials remain secure if the data in one of the servers is compromised, as compromising both servers would lead to the database credentials being disclosed.

In the case that the client application is compromised and a set of credentials are disclosed (e.g. through a key logger that registers the credentials as they are typed in). A malicious user is now able to initiate a connection to the system and authenticate, being given access to the operations that the stolen credentials grant to the client application. However, the malicious user is restricted to the operations provided by the S-DiSACA and the credentials obtained do not allow him to connect directly to the database in any way. This is the worst-case scenario where some data may be disclosed, but this risk is unavoidable as an impersonated client cannot be distinguished from a legitimate client.

If the authentication server is compromised instead, the malicious user has access to the usernames, hashed passwords, salts and the encrypted keys *C* for each application. With this information, the malicious user is capable of running brute-force attacks on the hashed passwords to crack them to impersonate legitimate users, but the same restrictions as the previous case still apply. Finally, the encrypted keys *C* are useless as they do not contain any information regarding the database credentials. They are randomly generated and encrypted with the Policy Manager public key P_{pub}, meaning that the Policy Manager is the only entity able to decrypt them anyway.

Finally, in the case where the Policy Manager itself is compromised, the malicious user would have access to the database credentials encrypted with the client symmetric keys *C*, which are randomly generated. Thus, attempting to crack the keys is almost impossible given large enough keys, not to mention that the database credentials can also be random.

The private keys must remain secure even if one of the servers is compromised to prevent their impersonation. Impersonating the Policy Manager, for example, would indirectly compromise the authentication server as well, as it sends information that only the legitimate Policy Manager should be able to access. This information includes the symmetric keys *C* sent during a client connection process, which would allow a malicious user to eventually decrypt database credentials. This can be detected by monitoring the connection attempts made to the authentication server and the Policy Manager,

among other techniques. If any mismatch in the information or missing requests are detected, then corrective action should be taken immediately.

Another approach is to provide the authentication service and the Policy Manager with signed digital certificates that identify the IP addresses authorized to use those keys. These digital certificates could be signed using an in-house certificate authority to avoid placing trust on an external entity, however, the client application should not be provided with one for authentication purposes. As argued before, these client applications can run on semi-public spaces which could lead to the certificates being stolen or replaced, thus the approach proposed here.

4.3.2 Communication Security

An encrypted communication channel could be easily implemented using Secure Sockets Layer (SSL)/TLS and digital certificates. As mentioned in the previous section, the authentication service and the Policy Manager could both be given a digital certificate signed by an in-house certificate authority, which would allow an SSL/TLS communication channel to be used.

However, this approach would only authenticate the servers, not to mention that it could be possible for a malicious user to install a new certificate authority on the client application since they can run in semi-public spaces. By installing one such authority, a malicious user could easily create his own certificates stating that they are the legitimate server and impersonate them. Attempting to use a digital certificate on the client applications to authenticate them is a very bad idea, as these certificates could be easily stolen for the same reason a new certificate authority could be installed.

Still, the SSL/TLS protocol to establish a secure communication channel is a good starting point. Versions of the SSL/TLS protocol that use pre-shared keys instead of digital certificates are been proposed[111], but they remain unavailable in most programming languages. Fig. 4.12 shows the TLS protocol messages exchanged between a client and a server when using certificates.

In broad strokes, the client and the server begin by exchanging "Hello" messages, which includes the TLS version and the cyphers supported. Then the server sends the public certificate with its public key to the client, which it uses to encrypt a master key S. Then both parties perform a "ChangeCipherSpec" operation, which creates a set of read and write keys from this key S. After that, data can be transmitted between them encrypted.

However, it is possible to establish an SSL/TLS protocol without using certificates. This is achieved by using the Diffie-Hellman key exchange protocol [112] in anonymous mode. This approach ensures that one key is generated and shared between the client and the server without network eavesdroppers being able to know that key. However, neither the client nor the server is authenticated in anonymous mode, potentially leading to man-in-the-middle attacks.

To resolve this issue, a key modification procedure is employed that not only prevents man-in-the-middle attacks but also authenticates the client and the server. If the pre-shared key between the client and the server is PSK and the key agreed through Diffie-Hellman is S, then the new secret S' used to encrypt the communication is calculated according to Eq. 4.17. Where H is a hash function and F alters the pre-shared key with some random value, like a salt, to prevent the application of rainbow tables [113].

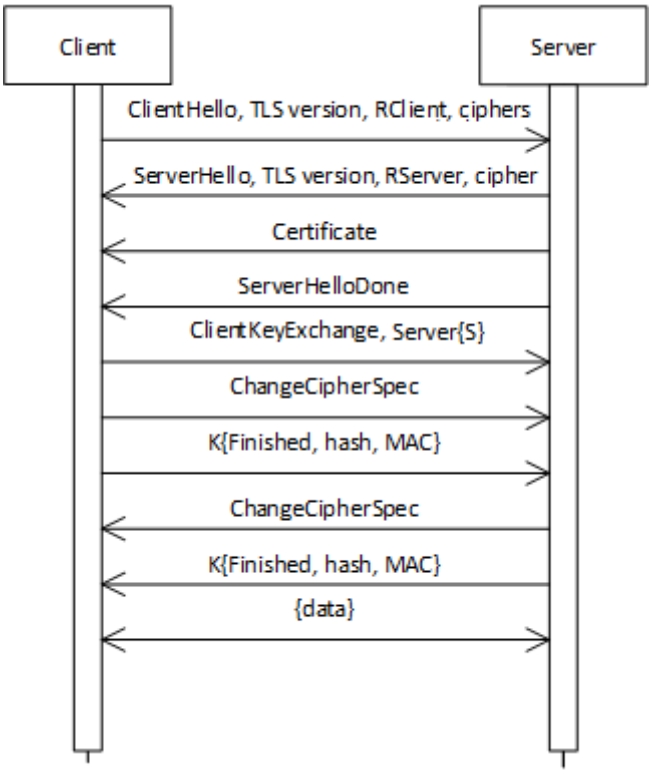


FIGURE 4.12: TLS protocol diagram using certificates[38].

$$S' = H(S + H(F(PSK))) \tag{4.17}$$

Since only the client and the server are supposed to know the pre-shared key PSK, if they can communicate after the agreed key S is modified to S' then they are authenticated. In the case of a man-in-the-middle attack, since the pre-shared key PSK is unknown to the attacker, all that is visible is the encrypted communication.

4.4 Summary

In this chapter, the changes made to the S-DRACA to enhance its security features were presented and bundled into a new architecture called S-DiSACA.

Approaches to generalize and protect the operations being executed on the data, along with some of the parameters required to execute them, were proposed and discussed to address RQ1. Moreover, the overall security of the architecture was discussed, including a new approach to protect the database credentials from being stolen on the client applications and alternative ways to establish encrypted network communication channels, addressing RQ2 and some of RQ3. Lastly, the final overall architecture of the S-DiSACA was presented and its life-cycle discussed, along with the new

Business Schema interface generation, implementation and usage procedures. This addressed the remaining aspects of RQ3.

However, some research questions remain, such as how to build flexible access control policies that can be modified at runtime to widen the application scenarios for this system (RQ4) and how can these systems be audited for correctness when users are loosely mapped to access control permissions (RQ5). The following chapter discusses and proposes solutions to these points.

Chapter 5

Fuzzy Access Control Decision Making

In this chapter, the research performed to integrate soft access control requirements into access control policies and how can these policies be enforced on an access control system.

To achieve this goal, an inference system capable of handling soft rules was studied and adapted into a decision making inference system that was integrated into the Secure, Dynamic and Distributed Soft Access Control Architecture (S-DiSACA) as its Policy Decision Point (PDP) (see Fig. 6.1), allowing the architecture to support application scenarios where the access control rules are hard to define clearly. To not tie this inference system to any application scenario, the concepts and rules used are defined via policy files, thus allowing to use it in any context provided that a policy has been created. This approach also makes it easier to integrate changes made to the access control requirements since only the policy files need to be modified instead of the access control system itself.

Several soft inference systems were evaluated [23], [41]–[44], [76]–[78], but two quickly showed to be the potential best candidates for adaptation and integration into an access control system, which were compared in section 2.2.3: the Mamdani-type Fuzzy Inference System (FIS) [45]; and the Sugeno-type FIS [46]. The reason that these two approaches are the best candidates for adaptation is that they are generic systems, whereas all the others specialize to deal with some specific problem. Moreover, both of these inference systems are usually distributed with fuzzy inference libraries while the others are not. Between the Mamdani and the Sugeno types, the Mamdani was chosen to be adapted since it has more widespread use and its output layer can be easily modified to produce binary outputs.

The application of this type of logic in access control comes with a steep price, which is the higher system auditing complexity. With this logic, it is difficult to determine beforehand which combinations of input variable values lead to a grant or a deny decision for a given request. Therefore, auditing policies based on this type of logic requires an exhaustive search over the input domain to ensure its correctness, which is problematic if the input variables have large domain sizes. This fact makes auditing difficult, and in turn, makes choosing this type of access control system less desirable. Here is where the biggest contribution to the scientific community is made with this thesis: an algorithm based on novel optimization techniques capable of enabling a security expert to audit the access control policies based on fuzzy logic is provided.

Therefore, this chapter discusses how the Mamdani-type FIS can be modified to produce binary outputs, analyses the approach for potential problems, and discusses the security considerations that must be made when integrating soft requirements into access control systems. It is divided as follows: section 5.1 discusses the changes made to the Mamdani-type FIS that led to the Binary Decision Fuzzy Inference System (BDFIS); section 5.2 discusses the problem of auditing the correctness of access control policies for BDFIS and presents the auditing algorithm; section 5.3 discusses the security considerations one must make when using BDFIS in an access control system; and section 5.4 summarizes the chapter.

5.1 Binary Decision FIS

A Mamdani-type FIS goes generally through six steps to produce an output from a set of input values. From these, one can determine that the output membership functions have a direct bearing on the output value, as they are truncated using the rule strengths and then combined to calculate a crisp output using a defuzzification method. Thus, these steps are evaluated in terms of what must be changed to support binary (i.e. grant/deny) outputs.

5.1.1 Fuzzy Rule Determination

The first step involves the determination of the fuzzy rules to be used by the FIS. The fuzzy rules use the input linguistic terms and their membership degrees, and by applying fuzzy logic a rule strength can be computed. This rule strength is then applied to an output membership function.

Since the idea is to have the FIS generate a binary output, the input linguistic terms can remain effectively the same. This allows any attribute to be used as an input and does not narrow the applicability scope of the BDFIS. This approach contrasts with some of the FIS used in some of the related work, where the input parameters are set *a-priori*. The output variables and their linguistic terms, however, have a direct bearing on the output domain. When it comes to an access control system, the output must be to either grant or deny a permission to a resource. To do this with BDFIS, each permission is defined as an output decision according to Def. 5.1.

Definition 5.1. Access permissions in a BDFIS are output variables associated with exactly two linguistic terms, called the fuzzy decision components (FDC): one for a positive decision FDC_+ (yes / grant); and one for a negative decision FDC_- (no / deny).

Not only does this allow permissions to be directly integrated into the fuzzy rules, but also to state which decision a given rule applies to each permission. Thus, a fuzzy rule written for the BDFIS takes the form "**if** A is LT_A **and/or** B is LT_B then Z is $FDC_\pm$", where A and B are input variables, LT_A and LT_B respective linguistic terms, and Z is an output permission variable. For example, "**if** $Expertise$ is $High$ **and** $Activity$ is $Moderate$ then $Read$ is $Granted$".

This example shows how BDFIS can be used to easily define soft access conditions: given a set of vague concepts about a subject (e.g. $Expertise$, level of $Activity$, etc.), the permissions ($Read$, $Write$,

etc.) to the resources are defined as output variables that are either granted (FDC_+) or denied (FDC_-) by each rule. How each rule influences the final decision is discussed next.

5.1.2 Input Fuzzification and Rule Strength

Before the fuzzy rules can be applied to calculate the rule strengths, the input variables need to be fuzzified. Fuzzification is the process that qualifies the input variables in terms of the defined linguistic terms. Thus, given a set of linguistic terms T_i for an input variable i, the membership degree of i to each linguistic term $t \in T_i$ is obtained the associated membership function μ_t as shown in Eq. 5.1:

$$\mu_t(i) : t \rightarrow [0, 1], t \in T_i \tag{5.1}$$

The membership functions μ should be defined by an expert in the context of the application scenario in which the BDFIS is used, since soft requirements such as the *Expertise* of a subject can change depending on the context.

After the membership degree is calculated for each of the linguistic terms, the fuzzy rules can be applied to determine the rule strength associated with each FDC according to Def. 5.2.

Definition 5.2. Given a set of fuzzy rules R, the rule strength of each rule $r \in R$ is determined by applying the fuzzy logic operators AND, OR, and NOT to the membership degrees of the input linguistic terms used in it. When more than one rule applies to the same FDC, the combined rule strength can be calculated by applying the OR fuzzy operator over every individual rule strength that pertains to it.

Thus, the fuzzification process of the input variables into membership degrees for each of the defined linguistic terms requires simply the application of the membership function associated with it. Then, these values are processed using the logic operators defined in the fuzzy rules, following, for example, the Łukasiewicz-Tarski logic, to generate the rule strength for each rule. Finally, in the case that more than one rule applies to the same output permission variable and FDC, the rule strength of each of these rules are combined using the OR fuzzy logic operator.

With the rule strengths associated with each output linguistic term calculated, the process of determining the consequence of each output permission variable can take place.

5.1.3 Consequence Determination

The consequences of an output permission variable are obtained simply by truncating the membership functions of each output linguistic term with the rule strength that was calculated on the last step.

Since it is intended for the output of the BDFIS to be binary, the FDC membership functions can be predefined to encode both a *Deny* and a *Grant* values. By pre-defining these membership functions, the behaviour of the BDFIS is the same regardless of the context it is used in (i.e. the output domain of the system is the same). The question is which membership functions to use for the FDC_+ and FDC_- linguistic terms.

Since these functions need to be truncated and later combined to be defuzzified, a way to simplify the process is to assign a singleton membership function. Def. 5.3 defines a singleton function.

Definition 5.3. A function $f(x)$ is called a singleton function if its output is 0 in its entire domain except for a single input value x_0, for which the output is 1 as shown in Eq. 5.2:

$$f(x) = \begin{cases} 1, & \text{if } x = x_0 \\ 0, & \text{if } x \neq x_0 \end{cases} \tag{5.2}$$

Thus, a singleton function allows assigning each FDC a single output value. The value chosen for the FDC_- and FDC_+ was 0 and 1, respectively. The reason why these particular values were chosen is that they help to simplify the defuzzification process even further, a process that is explained in the next subsection.

The membership functions for the FDC_- ($\mu_-(x)$) and the FDC_+ ($\mu_+(x)$) are then singleton functions as shown in Eq. 5.3 and 5.4, respectively.

$$\mu_-(x) = \begin{cases} 1, & \text{if } x = 0 \\ 0, & \text{if } x \neq 0 \end{cases} \tag{5.3}$$

$$\mu_+(x) = \begin{cases} 1, & \text{if } x = 1 \\ 0, & \text{if } x \neq 1 \end{cases} \tag{5.4}$$

Having defined the membership function for both FDC_+ and FDC_- the process of determining the consequence can now be explained. The consequence is calculated through a truncation process, which is explained in Def. 5.4.

Definition 5.4. The process of truncating a given function f at the value $y = y_0$ generates a new function g that has the same output as f except that any output value greater than y_0 becomes y_0, as shown in Eq. 5.5.

$$g(x) = min(f(x), y_0) \tag{5.5}$$

Since singleton functions only have one input value ($x = x_0$) where the output is not 0, then only one value needs to be truncated. Furthermore, the FDC membership functions μ_- and μ_+ are defined to output 1 on $x = 0$ and $x = 1$, respectively, and the rule strengths always lie in the $[0, 1]$ range due to being the result of the application of fuzzy logic. The consequence function C, then, is determined simply by replacing the output value 1 in $\mu_\pm$ by the rule strength $RS_\pm$ that applies to their respective $FDC_\pm$, shown in Eq. 5.6 and 5.7.

$$C_-(x) = \begin{cases} RS_-, & \text{if } x = 0 \\ 0, & \text{if } x \neq 0 \end{cases} \tag{5.6}$$

$$C_+(x) = \begin{cases} RS_+, & \text{if } x = 1 \\ 0, & \text{if } x \neq 1 \end{cases} \tag{5.7}$$

These consequence functions can then be used in the defuzzification step to generate a single, crisp output value. This simple value replacement explains in part why the usage of singleton functions simplifies the computation of the final decisions. However, further benefits are shown to be obtained during defuzzification in the next subsection.

5.1.4 Consequence Combination and Defuzzification

The next step is the consequence combination, which involves combining both the C_- and C_+ functions into an output distribution function θ for each output permission variable. This step is not always required in a Mamdani-type FIS, and as such, it is also not required in a BDFIS. This allows a system that uses the BDFIS to read the output of the consequence functions and to apply a more complex decision-making strategy. For example, human intervention may be required when the rule strengths for both FDCs are close to one another. However, if a simple strategy supported by the BDFIS is enough, the system can let the BDFIS produce a single output per permission variable.

To output a single crisp value, the output distribution function θ is first calculated as defined in Def. 5.5.

Definition 5.5. Given the consequence functions C_- and C_+ of an output permission variable O, the output distribution function θ associated with O is the result of applying an accumulative function S, as shown in Eq. 5.8.

$$\theta(x) = S(C_-, C_+) \tag{5.8}$$

In the case of the BDFIS, and considering that the consequence functions of a given output permission variable Z are the C_- and C_+ shown in Eq. 5.6 and 5.7, respectively, then the resulting output distribution function θ_Z is given by Eq. 5.9:

$$\theta_Z(x) = \begin{cases} RS_-, & \text{if } x = 0 \\ RS_+, & \text{if } x = 1 \\ 0, & \text{if } x \neq 0 \wedge x \neq 1 \end{cases} \tag{5.9}$$

Once a single function for the output permission variable is obtained, a defuzzification method can be applied to extract a crisp output decision value. This is where the value 0 and 1 for the membership functions μ_- and μ_+ matter the most. For example, let us consider that the centre of gravity defuzzification algorithm, one of the most used, is applied to the above function θ. Since the functions involved are singletons the discreet version of the center of gravity for singletons (COGS) formula is used instead, which the general formula is given in Eq. 5.10.

$$COGS(\theta_Z) = \frac{\sum_x x * \theta_Z(x)}{\sum_x \theta_Z(x)} \tag{5.10}$$

Where each value x is a value for which the function θ_Z does not output a 0. Note that had singleton functions not be used the calculation of integrals would have been required, increasing the complexity of this step considerably. Moreover, since the x values that do not output the value 0 in θ_Z are $x = 0$ and $x = 1$, the previous equation can be expanded and simplified, as shown in Eq. 5.11.

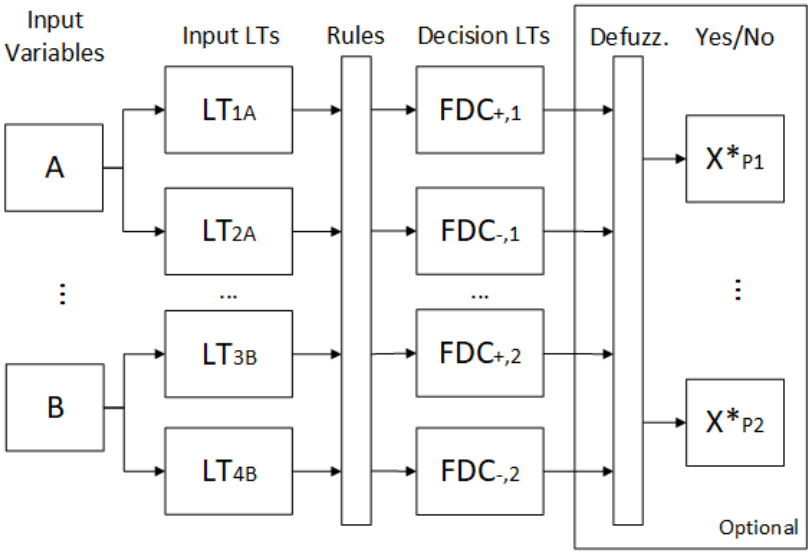


FIGURE 5.1: BDFIS conceptual block diagram[32].

$$COGS(\theta_Z) = \frac{0 * \theta_Z(0) + 1 * \theta_Z(1)}{\theta_Z(0) + \theta_Z(1)} = \frac{\theta_Z(1)}{\theta_Z(0) + \theta_Z(1)} \tag{5.11}$$

Therefore, with the careful selection of the FDC membership functions, the defuzzification step was simplified from having to calculate integrals to a simple division using two different values.

However, the above defuzzification method does not produce a 0 or 1 output, but some value that may be in-between as well. The closer that output value is to 0 the stronger the decision to deny the permission becomes, and likewise the closer it is to 1 the stronger the decision to grant the permission becomes. Here, several approaches can be used, such as applying a threshold δ (e.g. $\delta = 0.5$) to make a clear decision, or request the intervention of a human expert if the output is close to the threshold value. Any other method to derive a decision is valid, as well as using a defuzzification method that outputs a clear decision such as the first or last of maxima.

5.1.5 Conceptual Overview

This section presents a conceptual overview of everything that has been presented on the BDFIS so far.

Fig. 5.1 shows a diagram of the conceptual model of the BDFIS. Just like a Mamdani-type FIS, a list of crisp input values (A...B) is first fuzzified into its input linguistic terms (LT_{1A}, LT_{2A}, etc.) using the associated membership functions. The membership degrees to each of the linguistic terms are then used in a set of rules, which generate the rule strengths to be applied to each FDC ($FDC_{+,1}$, $FDC_{-,1}$, etc.) for permissions $P1$, $P2$, etc. These can be accessed directly by the system to make an access control decision, or they can be defuzzified first. This process applies a defuzzification method to output a crisp output value per permission X^*_{P1}, X^*_{P2}, etc.

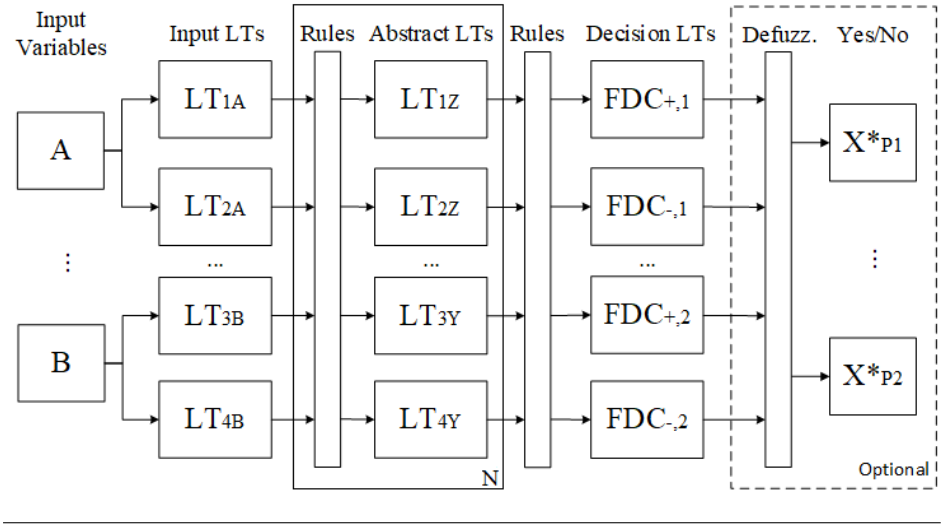


FIGURE 5.2: Type-N BDFIS conceptual block diagram[37].

From this diagram, it is clear that the BDFIS simply alters the output layer of the Mamdani-type FIS by pre-defining the output variables linguistic terms as FDCs and their membership functions as the singleton functions discussed in the previous subsection. However, while these changes allow the BDFIS to be easily used in contexts where decisions must be made from soft requirements, it was discovered that they also reduce the expressibility of the system as a whole. Since this approach requires the output decisions to be made from the input linguistic terms directly, it is is not possible to use more abstract concepts such as the *Expertise* of the user.

While this drawback may not impact simpler application scenarios, it can impact the interpretability of the fuzzy rules. Consider the rule "**if** *Number_of_Publications* is *High* **then** *Expertise* is *High*". A newly hired expert looking at this rule would immediately know what it is expressing: the expertise of the user. However, the BDFIS would require its rules to look like "**if** *Number_of_Publications* is *High* **then** *Write* is *Granted*". The concept of *Expertise* is implied in this case, and may be more challenging to interpret from this point of view.

Therefore, the idea of adding more layers to the BDFIS to support such expressibility was researched and is discussed in the next section.

5.1.6 Type-N BDFIS Abstraction

A problem with the expressibility of the BDFIS was mentioned where abstract concepts could not be defined and the input variable linguistic terms had to be used to make access control decisions directly.

A solution is proposed to allow BDFIS to support multiple abstract layers, which can be referenced by the name type-*N* BDFIS where *N* is the number of abstract layers. Thus, the BDFIS proposed until now was the type-0 BDFIS. The conceptual architecture of the type-*N* BDFIS is shown in Fig. 5.2.

As the figure illustrates, each new layer introduces a set of rules and abstract variables (along with their linguistic terms), forming a chain between the input and output layers of the BDFIS. Let us consider a type-1 BDFIS. With this type of BDFIS, it is now possible to define the *Expertise* of a subject mentioned in the previous section example within BDFIS, as shown in Listing 5.1.

LISTING 5.1: Example type-1 BDFIS abstract layer rules.

```
IF NoP IS Low  AND NoC IS Low  THEN Expertise IS Low
IF NoP IS Low  AND NoC IS High THEN Expertise IS High
IF NoP IS High AND NoC IS Low  THEN Expertise IS Medium
IF NoP IS High AND NoC IS High THEN Expertise IS High
```

This listing shows some example rules that may exist in an abstract layer that defines the abstract variable *Expertise* from the *Number_of_Publications* (NoP) and *Number_of_Citations* (NoC) variables. The process is the same as before, the rule strengths are calculated to each linguistic term, except now they are the linguistic terms of abstract variables, and the consequence combination and defuzzification processes do not take place. Instead, the rule strength for each abstract linguistic term is used directly in the next set of rules that now apply to the output FDCs, as shown in Listing 5.2.

LISTING 5.2: Example type-1 BDFIS output layer rules.

```
IF Expertise IS Low    AND Activity IS Low  THEN Write IS Deny
IF Expertise IS Low    AND Activity IS High THEN Write IS Deny
IF Expertise IS Medium AND Activity IS Low  THEN Write IS Deny
IF Expertise IS Medium AND Activity IS High THEN Write IS Grant
IF Expertise IS High   AND Activity IS Low  THEN Write IS Grant
IF Expertise IS High   AND Activity IS High THEN Write IS Grant
```

In this example, the *Expertise* and the *Activity* abstract variables are used to grant / deny the *Write* permission. At this stage, the rule strengths can be calculated and the same process of determining and combining the consequence functions for defuzzification can take place. However, these two sets of rules make it very clear what concepts are being used and how they impact the output decision. The input variables are fuzzified and used to determine the *Expertise* and the level of *Activity* of the subject, which in turn can grant the *Write* permission if they are high enough.

Moreover, if additional concepts are required that depend on previously defined abstract variables, then additional abstract layers can be used to define them.

5.1.7 BDFIS Policy Definition

So far, the BDFIS and its type-*N* generalization were presented and discussed. However, the policies that are going to be defined for this system have yet to be considered.

As mentioned in the previous chapter, the S-DiSACA PDP can access a policy data store, called the Policy Server, where policies are stored and that an administration component can manage. These policies can be modified at runtime, meaning that the BDFIS must be capable of reloading the policies and rebuilding its internal state with possibly new variables, linguistic terms and rules.

Since the BDFIS follows the well known Mamdani-type FIS, it was natural to research the existing solutions in terms of how Mamdani-type FIS can be defined and implemented. A standard language exists that defines these systems, called Fuzzy Control Language (FCL) [114]. Thus, BDFIS policies are defined using this standard language, an example of which is shown in Appendix A.

Since an FIS can be automatically generated from these files, upon a user request the S-DiSACA queries the Policy Server for the FCL policy files, implements them as BDFIS, and then executes them using the user parameters. Since each BDFIS outputs decisions for the permissions they are responsible for, the results can be cached for that user. After a certain amount of time, the policy files are retrieved again, the associated BDFIS reimplemented, and then executed to update the cache.

This approach can be used even if the access control decision-making engine is not a BDFIS but some other model, allowing the S-DiSACA to have flexible policies. When a policy is changed and a client application receives the associated error message, it can warn that it needs to be updated. A developer of the application, on the other hand, compiles the application using the new Business Schemas. Since the functions available in the Business Schemas are only those that were authorized by the PDP, the developer quickly knows what changed via compilation errors or Integrated Development Environment (IDE) hints and shorten the development time.

5.2 Policy Correctness Auditing

So far, an FIS capable of generating binary outputs was proposed to handle the access control decisions in scenarios that possess soft access control requirements. However, one of the desired features of access control that has not been discussed is permission auditing, which comprises of reviewing the capabilities of subjects and their permissions over resources [115]. This is also known as "before the act audit" and it is one of the most important features found in access control models such as Role-Based Access Control (RBAC).

As previously mentioned this application of fuzzy logic in access control comes with higher system auditing complexity. Since these systems use fuzzy logic, it is difficult to determine beforehand which combinations of input variable values lead to a grant or a deny decision for a given request. Auditing the policies requires an exhaustive search over the input domain to ensure its correctness, which is problematic if it has several input variables with large domains. This fact makes auditing difficult, and in turn, makes choosing this type of access control system less desirable.

Three-dimensional graphs of the output of a fuzzy system given two of the input variables are sometimes used in the literature to show the relation between them, such as in [43]. However, these graphs usually have low-granularity, even with relatively small input variable domains. Furthermore, these graphs do not take into account the local minima and maxima of the membership functions. This can translate into missed output changes in binary systems, with special importance for access control auditing.

Thus, in this section, optimization techniques for auditing BDFIS and other fuzzy systems that rely on binary outputs are proposed [37]. These techniques aim to reduce the search space to exhaustively verify the correctness of the policies and counter the aforementioned issues.

5.2.1 The Auditing Problem

To better explain the increased complexity in auditing an access control system based on access control, the non-trivial issues that must be addressed are discussed in this subsection.

In a crisp access control model, the input variables can be used directly in the rules and the way they affect the output is very clear. For example, "if *age* is greater than 18 then read is granted" clearly shows that if the subject is over 18 years old, then he is granted the read permission over some resource. Moreover, the 18-year-old condition is either completely true or false for any given subject.

While these models are used in scenarios where the list of subjects is known and permissions can be mapped explicitly, fuzzy models can be used to enforce access control in scenarios where permissions need to be associated with subjects at runtime. This process is carried out using the subject's attributes and the existing access control policies, determining how true certain statements about the subject are (e.g. the subject's expertise is high) and determining the subject's permissions from these statements. Since the subject attribute values can vary even between access attempts for the same subject, any permission auditing mechanism must determine the input variable values that lead to either a granted or denied access decision.

However, the impact of each input variable (i.e. the subject attributes) on the output decision is dependant on the other variables since they are all fuzzified and used in fuzzy rules. Consider the following set of five fuzzy rules, which calculates the level of *Expertise* based on the number of publications (NoP) and the number of citations (NoC). A *Write* permission is then granted or denied based on the level of Expertise.

1. **IF** NoP IS Low **THEN** Expertise IS Low
2. **IF** NoP IS High **AND** NoC IS Low **THEN** Expertise IS Medium
3. **IF** NoC IS High **THEN** Expertise IS High
4. **IF** Expertise IS Low **OR** Expertise IS Medium **THEN** Write IS Deny
5. **IF** Expertise IS High **THEN** Write IS Grant

Since fuzzy logic allows for partial truths, a subject can be somewhere in between a low and high number of publications and citations. This translates into different membership degrees associated with each linguistic term, which would activate every rule presented. Thus, the subject can have low, medium and high Expertise at the same time (with different degrees). Since the output functions for each linguistic term are added together to determine the final output, even if only the number of citations is increased to add strength to the high expertise linguistic term (through rule 3) the number of publications influences the decision turning point by adding strength to the low and medium expertise linguistic terms (rules 1 and 2).

Therefore, a naïve approach to accurately determine which combinations of input variables lead to a granted or denied decision is to perform a brute-force search over the entire domain. A brute-force approach is, obviously, not scalable to a big number of variables. If the average size of the domain for a variable D is $|D|$ and there are N variables, then the search space approximates $|D|^N$, which

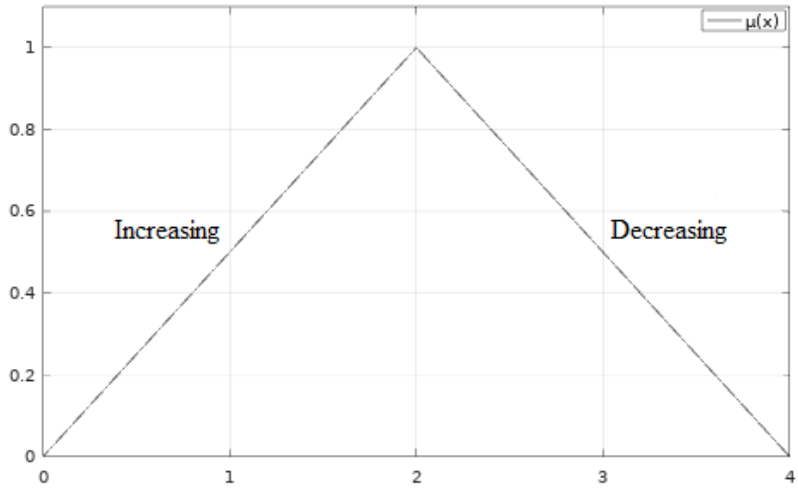


FIGURE 5.3: Example increasing and decreasing function.

grows exponentially. Moreover, searching the entire domain without considering the local minima and maxima the membership functions can lead to missed changes in the output decision.

To solve this problem a careful analysis of the linguistic term membership functions together with the fuzzy rules where each term appears in is required. An initial optimization procedure to reduce the search space was researched, and several useful properties of the FIS were found that do allow for this optimization to take place. The following sections present this analysis and incorporate its results as a pre-processing step into an optimized auditing algorithm.

5.2.2 Theorems and Definitions

Before the techniques used to optimized the search space can be presented, the theorems and definitions used to development must be introduced and discussed. Since the main problem being faced is the difficulty in interpreting how changes to the input variables affect the output decisions, the following assumptions are made:

1. Only one variable is updated at a time between permission evaluations, and the update cannot move from an increasing portion of the membership function to a decreasing portion or vice versa without stopping at a point where the derivative is 0 or undefined.

2. The defuzzification method is consistent in the sense that if the "Grant" FDC rule strength increases more than the "Deny" FDC rule strength, then the final decision cannot change from "Grant" to "Deny" and vice versa.

Assumption 1 guarantees that when the search algorithm updates a variable, no permission changes go unchecked. To exemplify this point, consider Fig. 5.3 which shows a very simple membership function. In the range $0 < x < 2$, the function is increasing, while in the range $2 < x < 4$

it is decreasing. Assumption 1 states that the algorithm cannot move from the first range to the second without passing by a point where the derivative is 0 or undefined. This point in the figure is $x = 2$ where the two ranges meet. This always represents a local maximum or minimum of the function, and if it is not checked a decision output change could be missed.

Assumption 2 is used to correlate changes in the input values with changes in the output decision. If the rule strength to the FDC_+ increases while the FDC_- decreases or remains the same, then it should not be possible for a decision to change from "Grant" to "Deny".

Thus, to optimize the search algorithm the membership functions should be partitioned into domain ranges that either increase, decrease or keep the membership degree constant. This process allows to classify how each membership function affects the output decision according to some domain range, and the points were these ranges meet include every local minima and maxima of the function. However, it is necessary to show first that any membership function can be partitioned into domain ranges where this is true, known as monotonic sub-domains.

Theorem 5.1. All membership functions are partitionable into monotonic subdomains.

Consider a membership function $\mu : D \to [0, 1]$ with domain $D \subseteq R$. A monotonic function is defined as a function where the signal of its derivative never changes signal, i.e. it is always non-decreasing or non-increasing.

If μ is continuous, it either is already monotonic for its entire domain or not. If it is not a monotonic function, then there exists a point x where its derivative $\mu'(x) = 0$ and the signal of $\mu'(x - \epsilon)$ and $\mu'(x + \epsilon)$ differ. These points are the local minima or maxima by definition. If the domain is partitioned at each point x where this occurs, then by definition each subdomain is monotonic.

If μ is not continuous, then there are discontinuity points. By partitioning the domain at each one of these discontinuity points they are removed from each subdomain, therefore each subdomain must be continuous. Since each subdomain is continuous, then they must also be partitionable into monotonic subdomains as shown above. QED

As stated in Theorem 5.1, any membership function can be divided into monotonic subdomains. In the extreme case where the function is discontinuous at every value of its domain, then the result would be a set with every point of its domain. However, this is not a common occurrence as membership functions tend to model natural concepts and thus are usually defined using normal distribution functions, piecewise linear functions or other continuous functions.

Having proved that membership functions can always be partitioned into monotonic subdomains, the question now is how an input variable value can be updated to check for output decision changes while ensuring that assumption 1 remains true. Def. 5.6 introduces how this can be achieved.

Definition 5.6. Consider an input variable A with a current fixed value x_n that needs to be updated to the next value x_{n+1} and a set of membership functions U_A, one for each linguistic term. The update must guarantee that x_{n+1} is within the same monotonic domain range as x_n for each $\mu_A \in U_A$.

From this definition comes that each point that divides two different monotonic domain ranges must always be checked when transitioning between them since these are the only points that are considered to be in both. Thus, given that each range is monotonic, assumption 1 is always satisfied.

Linguistic Term Contribution

Having discussed how each variable must be updated to yield consistent results, the question now lies in how each monotonic sub-domain partition affects the output. To answer that question, each linguistic term must be classified in terms of which rules they are used in, and which FDC these rules affect as consequences.

Definition 5.7. A bipolar linguistic term is a linguistic term that exists in at least two rules whose outcomes can make the rule strengths of both FDCs change in the same direction (increasing or decreasing).

A bipolar linguistic term, then, has the capability of increasing or decreasing both the "Grant" and "Deny" FDC rule strengths for a given permission at the same time. Therefore, any linguistic term that is non-bipolar either does not change the FDC rule strengths, changes just one, or changes both in opposite directions.

LISTING 5.3: Set of example binary output rules.

IF A IS A_1 **AND** B IS B_1 **THEN** P IS Grant
IF A IS A_1 **AND** B IS B_2 **THEN** P IS Deny

To exemplify this, consider the rules shown on Listing 5.3. Two input variables were used, A and B, plus one single output variable P. Each variable also has several linguistic terms: A_1, B_1, B_2, *Grant*, and *Deny*. It is possible to see that the linguistic terms A_1 and B_1 are used to determine the rule strength to the Grant FDC, while the linguistic terms A_1 and B_2 are used to determine the rule strength to the Deny FDC. The Grant and Deny FDCs belong to the output variable P.

Thus, the linguistic terms B_1 is only used to determine the rule strength to a single FDC (Grant). Likewise, linguistic terms B_2 is only used to determine the rule strength to a single FDC (Deny). The linguistic term A_1, however, is used in two rules that apply to opposite FDCs of the same output variable.

Given this scenario, A_1 is a bipolar linguistic term, as it affects the rule strength to both the Grant and Deny FDCs. Since this does not happen for the linguistic terms B_1 and B_2 they are considered non-bipolar.

Knowing that a linguistic term is bipolar or non-bipolar, the question now is how these affect the output decisions of the system when the associated variable is updated. Instead of tackling how every linguistic term affects the output as a whole, the contribution of each linguistic term is defined across its domain first.

Definition 5.8. Consider a non-bipolar linguistic term, its associated input variable A and membership function μ_A. When x_n is updated to x_{n+1} it is defined that:

- If $\mu_A(x_n) - \mu_A(x_{n+1}) < 0$ (i.e. a negative discreet derivative), then the contribution of the linguistic term increases.
- If $\mu_A(x_n) - \mu_A(x_{n+1}) > 0$ (i.e. a positive discreet derivative), then the contribution of the linguistic term decreases.

- If $\mu_A(x_n) - \mu_A(x_{n+1}) = 0$ (i.e. a discreet derivative equal to 0), then the linguistic term has no contribution.

However, knowing how the contribution changes for a given update is not enough, as the membership function can support the "Grant" and/or the "Deny" FDC. Def. 5.9 shows how to determine which FDC is supported by a positive, negative or no contribution update.

Definition 5.9. Consider a non-bipolar linguistic term. It is said that updating the value of the associated variable supports a specific FDC if:

- The rule strength increases for that FDC.
- Or the rule strength decreases for the opposite FDC.

These two situations are the same in terms of the overall effect on the output decision change, hence both being classified as the linguistic term supporting the same FDC. However, an update must change the input value associated with a variable, and each variable may possess more than one linguistic term. Therefore, the linguistic terms must all be considered to determine how a variable update ultimately affects the output.

Variable Update Support

Having defined how each domain range is classified and how it affects each FDC, it is now possible to start proposing and proving some theorems on how updating a variable within these domain ranges can affect the output.

Theorem 5.2. If the variable A, at a current value x, is updated and all its linguistic terms have no contribution, then the update does not change the final decision.

Consider a rule $R : I^n \mapsto FDC$ that takes a set of antecedent linguistic terms I and outputs the value for the consequent FDC.

Using Assumption 1, it is known that only one variable can be updated at a given time. All other variables are then constants.

From Def. 5.8 comes that a linguistic term with no contribution does not change its rule strength output from the update to the input variable A. Therefore, if all linguistic terms for the variable A have no contribution to the update, then the output is constant.

Since all rule strengths are constant for the update, there can be no change in the FDC rule strengths and therefore the final decision must be the same. QED

Theorem 5.3. If the variable A, at a current value x, is being updated and all its linguistic terms support the same FDC or have no contribution, then the update either changes the final decision to the FDC being supported or does not change it.

First, the linguistic terms that have no contribution were demonstrated to not change the final decision with Theorem 5.2.

Regarding the linguistic terms that support the same FDC, they change the rule strength output of the rules in two different ways:

- They increase the rule strength in rules that support the specified FDC.
- Or they decrease the rule strength in rules that support the opposite FDC.

Since all other linguistic terms from other variables do not change and are therefore constant in the update, the rule strength either increases for the specified FDC, decreases for the opposite FDC or remains the same due to some constant being greater/lesser than the updated membership degree, depending on the minimum/maximum functions.

Therefore, the gap between the "Grant" and "Deny" FDC rule strengths either remains the same, the rule strength to the specified FDC increases or the rule strength to the opposite FDC decreases. Due to Assumption 2 and since the FDC with the greatest rule strength sets the output decision, the decision can either change from the opposite FDC to the specified FDC or not change at all. QED

This result implies that if the decision made is already the supported FDC, then the update cannot change the decision. This also means that if the decisions are known for the entire domain of one variable A and another variable B is updated in which all its linguistic terms support a specific FDC, then the ranges for which variable A had a decision made in favour of the specified FDC do not need to be reevaluated.

Thus, each input variable should have its domain partitioned into ranges where all linguistic term membership functions have no contribution, or all support the same FDC. Ranges where none of these occur are considered to have an unknown contribution to the decision-making process and must be checked. However, the designed algorithm makes the most out of this partitioning, a process that is explained in the next section.

Put together, Theorem 5.2 and Theorem 5.3 form the basis for the optimization of the search space, which is identified and built into a search algorithm where entire ranges from each variable domain can be skipped.

5.2.3 Algorithm Techniques

In this subsection, the designed algorithm techniques are presented. These include the method by which the input variables have their domain partitioned and an analysis of how each partition reduces the domain range.

Range Partitioning

The process of variable range partitioning is an important one for the implementation of the optimized search algorithm and is detailed in this section. In short, a partition $[a, b]$ must possess a consistent FDC contribution that allows the algorithm to determine if the domain values in the partition interval must be evaluated or can be skipped.

Figure 5.4 shows an example input variable membership functions for the linguistic terms 1, 2 and 3. The domain ranges from 0 to 100 on the x-axis, while the y-axis shows the membership degrees that a given input value has to each linguistic term.

Per Theorem 5.2, a variable has no contribution (NC) on a given value if all LTs have a discreet derivative equal to 0. The discreet derivative is equal to 0 if the function is constant, which means that

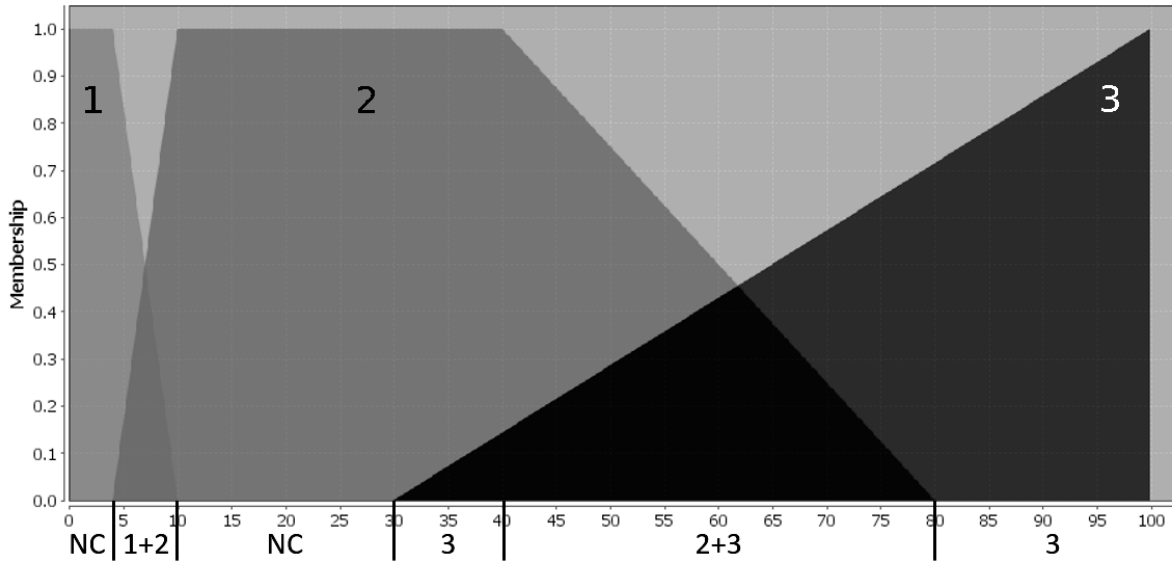


FIGURE 5.4: Input variable domain partitioning[37].

the value ranges $[0, 4]$ and $[10, 30]$ have no contribution to any decision. Therefore, once the decision is known for a single value within each range, then the decision for the other values is the same.

Then per Theorem 5.3, the information regarding the rest of the ranges depends on the contribution of the linguistic terms that do have some sort of contribution. This applies to the following ranges:

- $[4, 10]$: Depends on the linguistic terms 1 and 2, as linguistic term 3 has no contribution.
- $[30, 40]$: Depends solely on linguistic term 3, as both linguistic terms 1 and 2 have no contribution.
- $[40, 80]$: Depends on both linguistic terms 2 and 3. Linguistic term 1 has no contribution.
- $[80, 100]$: Depends solely on linguistic term 3, as both linguistic terms 1 and 2 have no contribution.

On these ranges, only when all the linguistic terms contribute to the same FDC at a given range is it possible to decide that a range contributes to one FDC. Otherwise, the contribution is unknown and the decision may change more than once.

To explain why the contribution is unknown, consider that linguistic terms 1 and 2 in the range $[4, 10]$ contribute to opposite FDCs and that they belong to the input variable A. Even when piecewise linear membership functions are used, where updating the input variable means that the rate at which the membership degree of the subject to the Granted and Denied FDCs changes is the same, other variables (with constant membership degrees) can be used within the defined rules, such as the input variable B in the rules shown in Table 5.4.

LISTING 5.4: Set of example binary output rules.

IF A IS A_1 **AND** B IS B_1 **THEN** P IS Grant
IF A IS A_2 **AND** B IS B_2 **THEN** P IS Deny

Consider that the fuzzy operator AND applies the minimum function to calculate the rule strength. While $\mu_{A_1} < \mu_{B_1}$ and $\mu_{A_2} < \mu_{B_2}$, any update to variable A affects the rule strength to both FDCs, which can lead to the final decision to change unexpectedly.

However, there may come a point where one of the rule strengths stops changing, such as μ_{A_1} or μ_{A_2} becoming greater than μ_{B_1} or μ_{B_2} respectively. In this case, if for example $\mu_{A_2} > \mu_{B_2}$ and the current decision output is "Deny", then rule strength from the second rule becomes constant (because B is constant and the AND operator takes the lowest of the two) and any further updates to variable A can only increase the contribution to the Grant FDC due to the first rule. This may eventually trigger the final decision to be changed to "Grant" as A is updated.

These final decision changes are hard to predict, as they depend on the current membership degrees of each linguistic term in each rule plus the rate at which each membership degree changes when a variable is updated. Adding more rules only compounds the problem. Therefore, since the decision may change several times, it is unknown how updating a variable in a region of unknown contribution affects the decision outcome. Finally, non-linear membership functions may be used, which increases the complexity of this problem by allowing the contribution to each FDC to change at different rates each time an input variable is updated.

Analysis and Variable Ordering

Since the algorithm must search every input variable, the easiest way to implement a search algorithm is via a recursive function. This implies that a variable has its entire domain checked before the next one is updated, and the impact of the variable order on the optimization efficacy is analyzed in this section. Finally, since the algorithm determines the decision output of the system each time a variable is updated, each decision is assumed to be stored and accessible in some way.

Consider the following definitions for a given policy:

- E is the number of calls to the evaluation engine required to analyze the policy;
- I is the number of input variables;
- $N_0(x)$, $N_1(x)$, and $N_2(x)$ are the number of separate ranges of no contribution, single contribution and unknown contribution of the x^{th} input variable, respectively;
- $R_0(x,n)$, $R_1(x,n)$, and $R_2(x,n)$ are the size of the n^{th} range of no contribution, single contribution and unknown contribution of the x^{th} input variable, respectively.

Given these definitions, the number of evaluation calls in the worst-case scenario (a brute-force search) is given by Eq. 5.12.

$$E = \prod_{x=1}^{I} \left[\sum_{n=1}^{N_2(x)} R_2(x,n) + \sum_{n=1}^{N_1(x)} R_1(x,n) + \sum_{n=1}^{N_0(x)} R_0(x,n) \right] \tag{5.12}$$

In this scenario, the number of calls to the evaluation engine E is the product of the size of the domain of each input variable. The domain size is obtained by taking the size of each separate range of no contribution, single contribution, and unknown contribution, and add them all together.

The first summation describes the ranges of unknown contribution, the second describes the ranges of single contribution and the third the ranges of no contribution. Each summation is analyzed in turn, to see how the proposed algorithm affects it.

Consider the third summation, as it describes the total size of the ranges of no contribution, can be removed almost in its entirety since these ranges do not change the decision output. However, the algorithm must evaluate at least one value from each of these ranges. Thus, the number of evaluation calls can be optimized as shown in Eq. 5.13.

$$E = \prod_{x=1}^{I} \left[\sum_{n=1}^{N_2(x)} R_2(x,n) + \sum_{n=1}^{N_1(x)} R_1(x,n) + N_0(x) \right] \tag{5.13}$$

Consider now the second summation, which describes the total size of the ranges of single contribution for each input variable. Since these ranges support only one of the two possible decision outputs, every time a variable is updated from a_n to a_{n+1} within one of these ranges it is possible to skip every value combination that used a_n and resulted in the decision supported by the range.

If *every* decision output matches the supported decision, then the rest of the range can be skipped. This means that every variable, including the first one, can have parts of its domain skipped by some factor $\beta_x, \beta_x \in [0,1]$, which can be different for each variable.

Furthermore, if the second variable also has ranges of single contribution, then the factor β_x of that variable and the ones that follow it can only grow closer to 1 since it adds to the range of values that can be skipped. Therefore, the β_x factors satisfy the following relation:

$$0 \leq \beta_1 \leq \beta_2 \leq \cdots \leq \beta_I \leq 1$$

Thus, Eq. 5.14 shows the new formula that generates the number of evaluation calls when applying all the shown optimizations.

$$E = \prod_{x=1}^{I} \left[\beta_x \left(\sum_{n=1}^{N_2(x)} R_2(x,n) + \sum_{n=1}^{N_1(x)} R_1(x,n) + N_0(x) \right) \right] \tag{5.14}$$

Since the single contribution ranges of the first variable affect all the following variables in the order, the variable with the largest size of single contribution ranges should be placed first. The variables should be ordered by the decreasing size of their single contribution ranges, as their impact

diminishes as they get closer to be the last variable (smaller ranges leave results in a smaller amount of values that can be skipped). This way, the β_x factors are maximized as much as possible early on.

Given this, the variables should also be ordered by the increasing size of their unknown contribution ranges. Since unknown contribution ranges (described by the first summation) cannot be optimized on their own, the largest ranges should be applied to the highest β_x factor possible to reduce their impact.

This can be problematic when the same variable has both the largest size of the unknown contribution and single contribution ranges. Thus, a possible heuristic to the ordering of the variables is to order them based on the normalized ratio of single contribution ranges to their unknown contribution ranges.

5.2.4 Integrated Overview

In this section, every step necessary to optimize the search space for the auditing algorithm is put together and shown from a holistic perspective. Fig. 5.5 illustrates this approach using a flow diagram, starting from the linguistic term analysis until the application of the algorithm itself.

There are two major flows: the pre-processing flow and the algorithm application flow. In the pre-processing flow, the first derivatives of the membership functions are analyzed to find the domain ranges where they are monotonic (Theorem 5.1). At the same time, the linguistic terms are classified by their bipolarity, as defined in Def. 5.7. These results are used to determine the contribution and support of each domain range in the next phase using Def. 5.8. If the function is constant then it has no contribution, and if it has a single contribution (because it is not bipolar) then the support is obtained by applying Def. 5.9. Otherwise, it must be bipolar and has an unknown contribution. Finally, the support of each variable update can be determined. First, the variable domain is partitioned according to the partitions of all its membership functions (as shown in Fig. 5.4). Then, Theorem 5.2 and 5.3 are applied to determine how every membership function in a given range affects the output when put together.

In the algorithm application flow, the data obtained from the pre-processing flow is used to optimize the domain search. The first variable is updated according to Def. 5.6 and then the support associated with the update is checked. If it supports no decision, then every decision evaluated in the previous update remains the same (Theorem 5.2). If it supports a single decision, then every case where that decision was outputted remains the same (Theorem 5.3). Otherwise, every output decision must be evaluated. The decision outputs for this update are saved, and if no more variables or updates are pending, the algorithm ends.

5.2.5 Type-N BDFIS Generalization

The optimizations shown so far were made for fuzzy systems that contain a single layer of rules and therefore the input linguistic terms are mapped directly to the output decision FDCs.

However, this is not the case for systems with more than one layer of rules, like the type-N BDFIS, where $N > 0$. In this case, there are N layers of abstract variables and linguistic terms between the

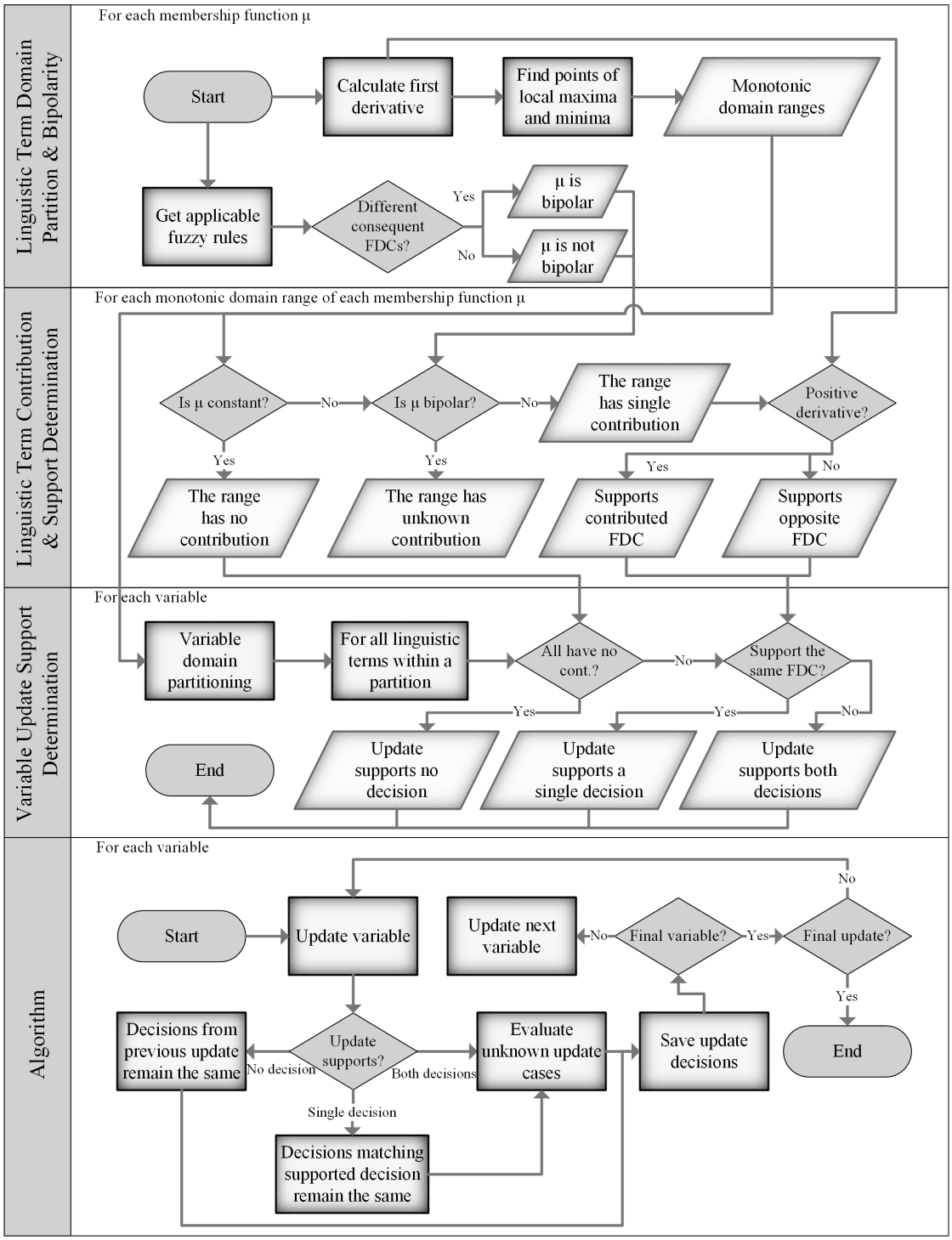


FIGURE 5.5: Algorithm flow diagram[37].

input variables and the output decision variables. This means that the input linguistic terms are mapped to the linguistic terms of the variables in the first abstract layer.

Thus, to determine the contribution of each input linguistic term it is necessary to know to which FDC the abstract linguistic terms on the first layer contribute towards.

Consider a type-N BDFIS, where there are N layers of abstract variables. Since the rules map them to the decision FDCs, the domain partitioning procedure described in chapter 5.2.3 could be applied to the last abstract layer to determine the contribution of each abstract linguistic term.

Then, on the layer $N-1$, the linguistic terms are mapped to another set of linguistic terms that have their contribution determined. The contribution of the linguistic terms of the variables in the layer $N-1$ can then be determined as detailed in Def. 5.10.

Definition 5.10. Consider the set of linguistic terms LT^n that are defined in the n^{th} layer of abstract variables. The FDC contribution of a linguistic term LT_i^n, $LT_i^n \in LT^n$ given the FDC contribution of the linguistic terms in LT^{n+1} is determined as follows:

- If all linguistic terms in LT^{n+1} that LT_i^n is mapped to have no contribution, then LT_i^n has no contribution.
- If at least one linguistic term in LT_{n+1} that LT_i^n is mapped to contributes to the same FDC and the rest have no contribution, then LT_i^n contributes to that FDC.
- If at least one linguistic term in LT_{n+1} that LT_i^n is mapped to has unknown contribution, then LT_i^n has unknown contribution.
- If at least two linguistic terms in LT_{n+1} that LT_i^n is mapped to contributes to different FDCs, then LT_i^n has unknown contribution.

Thus, by applying the above rules repeatedly to each layer from the output layer to the input layer, it is possible to determine the contribution of every linguistic term in the system including the input linguistic terms. This allows the optimizations found in the previous chapter to be applied once again.

5.3 Security Considerations

Until now, the methods by which soft access control requirements can be incorporated into an access control system such as the S-DiSACA has been explored. However, the very nature of the systems used to handle these requirements raise questions regarding their applicability to access control in the first place.

In the last section, techniques to optimize the domain search for a policy correctness algorithm were presented, and while it opens the door to future enhancements and additional optimizations, it also shows that these type of systems are not easy to fully control. Certain combinations of rules may grant or deny access to some unexpected users when certain decision outputs change briefly next to local minima and maxima of the membership functions.

Therefore, when deploying the S-DiSACA a security expert must evaluate what kind of decision-making engine (crisp or fuzzy-based) fits best a given application scenario. While the BDFIS presented

can define crisp access control requirements (by using singleton input membership functions) such as roles, a RBAC model may be better suited instead. Not only would that allow a security expert to quickly determine who has access to the resources, but a scenario where roles are used also tends to have a set of users that is known to the system at all times as well. These features increase the trust that the system always makes reliable decisions and should be used to protect highly sensitive information.

A fuzzy-based decision-making system like BDFIS, on the other hand, can grant and deny access to any user that attempts to access a resource as the mapping between users, permissions and resources are loosely defined with fuzzy rules. This approach is better deployed in application scenarios where the data is not very sensitive but can be accessed and modified by anyone, or in situations where users can go rogue. These scenarios include community-maintained data (such as Wikipedia) and the Internet of Things (IoT) where devices can malfunction or be exploited, altering their behaviour. The proposed policy correctness auditing techniques close a gap in the deployment of fuzzy-based access control systems in these scenarios, where it is now possible to analyse their expected behaviour and to validate the policies before deployment.

Another aspect to consider is the formal verification of the inference chains. As shown in this chapter, rules can conflict with each other in such a way that input variables can support different output decisions at the same time, leading to ranges of Unknown support that must be checked. Formal verification of these inference chains not only allows to verify if the set of rules in the policy are consistent with each other, but it also enhances the gains of the optimization techniques for the auditing algorithm by addressing the Unknown support ranges.

5.4 Summary

In this chapter, the possibility of supporting and incorporating soft access control requirement in the S-DiSACA was discussed and analysed, along with the applicability of such requirements in access control scenarios.

A binary decision FIS called BDFIS was developed that adapted the well-known and ubiquitous Mamdani-type FIS to produce binary outputs for access control purposes. Since FIS can be used to loosely map users to permissions based on policies defined on a database that can be modified at runtime, its development was able to address RQ4.

Regarding the policy correctness auditing of BDFIS, techniques based on theorems identified from these types of systems were researched and developed that allowed to reduce the domain space that needed to be searched to verify every input combination. These techniques can greatly reduce the time spent checking the situations in which permissions are granted or denied while guaranteeing that every decision change is found. This allows a security expert to evaluate the policy correctness and answers RQ5.

Finally, a short discussion of the implications of using fuzzy-based access control systems was also carried out to address RQ6. A proof of concept of the S-DiSACA integrated with a BDFIS as its PDP follows in the next chapter to demonstrate the features presented so far.

Chapter 6

Access Control System Architecture and Evaluation

In this chapter, a proof of concept[1] for the correctness and performance evaluation of every methodology and approaches mentioned so far are presented. The proof of concept of the Secure, Dynamic and Distributed Soft Access Control Architecture (S-DiSACA) uses Wikipedia article data and a custom application programming interface (API) that provides access to it, as well as several different policies. A client application was also developed, which shows the behaviour of the architecture in normal and abnormal contexts.

First, the implementation details of the architectural features discussed in chapter 4 are presented, alongside some deployment and usability cost observations. Second, the correctness evaluation focuses on testing the architecture in abnormal contexts and ensuring that the expected behaviour is observed, such as attempting to forcefully modify a parameter value obtained from a previous operation or attempting to subvert the defined sequence of operations. Finally, the performance assessment focuses on the amount of time that is required to realize each of the steps in the life-cycle of the architecture to authenticate, obtain the metadata, initialize the sequence controller, generate the business schemas, and to compile them. Furthermore, the time required to issue operation requests until they are served is also measured, along with the performance optimization gained in the policy correctness auditing algorithm with the proposed techniques.

This chapter is divided as follows: section 6.1 details the implementation of the architectural features; section 6.2 elaborates on the deployment and usability costs; section 6.3 demonstrates a client application and evaluates the correctness of the architecture; section 6.4 shows the performance assessment results; and lastly, section 6.5 compares the final architecture with the related works from the state of the art.

6.1 Access Control System Architecture Implementation

In this section, the S-DiSACA evolution and its life-cycle since the business schema interfaces are requested and generated until the application can execute operations over the data are presented.

[1] https://code.ua.pt/projects/s-draca/repository?rev=phd

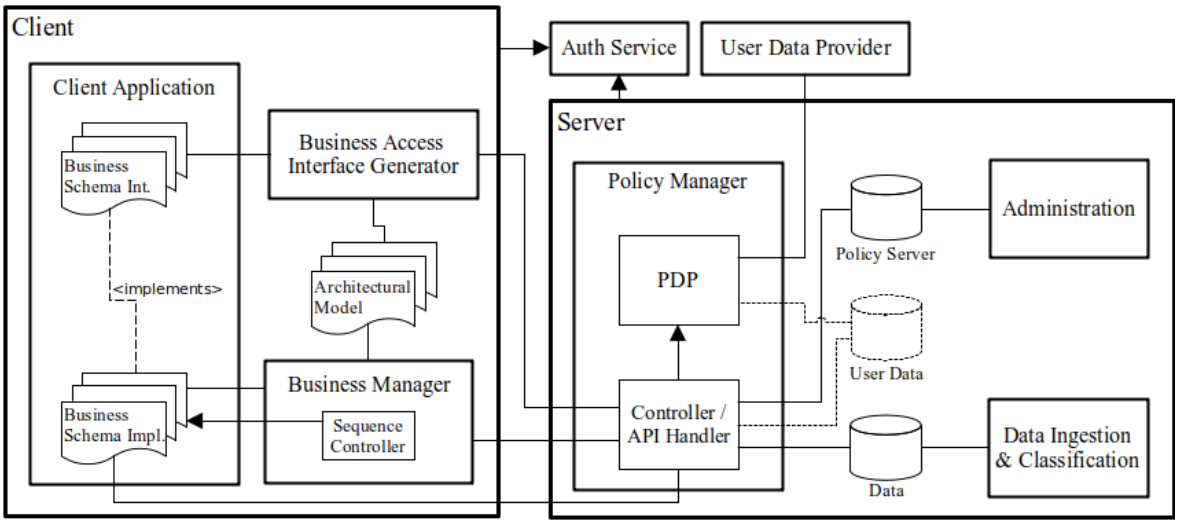


FIGURE 6.1: S-DiSACA overview.

Furthermore, the interface generation and implementation processes is also detailed for the new data access API.

Fig. 6.1 shows the new architecture, which is a direct evolution from the Secure, Dynamic and Distributed Role-based Access Control Architecture (S-DRACA) shown in Fig. 2.1 that includes the changes and new features presented in the previous chapters.

The most notable changes are on the server-side of the diagram, where a controller and API handler were added to the Policy Manager to account for the changes discussed for operation sequencing. An optional user data store was also introduced, which may contain the data about the users, as well as an external user data provider that can provide this data instead. The authentication service was also included to reflect the changes discussed in section 4.3, and a new Policy Decision Point (PDP) component was added to handle the access control decisions from the policies stored in the policy data store. The administration component allows manipulating the policies stored in the policy data store and the data ingestion & classification component allows to insert data into the system. One important aspect of this PDP is that it is now connected to a policy data store. This means that policies can now be changed through the administration component at runtime, and the PDP can automatically detect new policies and enforce them.

The following subsections detail how these components work together during the life-cycle of an application that uses the S-DiSACA.

6.1.1 Business Schema Interface Generation

The most important feature in the S-DiSACA is the automatic generation of the data access APIs called Business Schemas. Since these interfaces are tailored to each client application, a method is necessary to provide them with the interfaces for the development process.

When a client application being developed, a tool can be used to connect to the Policy Manager and to generate the Business Schema interfaces. It connects and authenticates using the procedures detailed in this chapter, and once that process is completed, the API metadata is requested.

The metadata received by the client application is shown in Listing 6.1 and is comprised of the flowcharts that it is authorized to execute ("sequences" JSON object) and the API function names, parameters and their data types ("interface" JSON object). The decision of whether a client application is authorized to execute a given flowchart is made by the PDP during the metadata generation process, using the access control policies defined in the Policy Server data store.

LISTING 6.1: API metadata JSON schema.

```
1  {
2    "interface" : [{
3        "name" : "op_name",
4        "returnType" : "<return_type>",
5        "modifiers" : "<modifierss>",
6        "parameters" : [{
7          "name" : "<param_name>",
8          "type" : "<param_type>",
9          "modifiers" : "<param_modifiers>"
10       }, ...]
11 }, ...],
12   "sequences" : {
13     "<sequence_name>" : {
14       "!entryPoint" : "<op_name>",
15       "<op_name>" : {
16         "cacheCode" : "<cache_code>",
17         "nextOps" : [
18           "<next_op_name_1>",
19           "<next_op_name_2>",
20           ...
21         ]
22       }
23     }
24   }
25 }
```

The "interface" JSON array is quite straightforward. It contains a list of JSON objects, one for each operation, that contains its name (line 3), a return type (line 4), modifiers (line 5) and parameters (lines 6-10). The "parameters" entry is also a JSON array of objects that contains the name, type, and modifiers of each parameter.

The "sequences" JSON object contains an object for each sequence indexed by its name (line 13). This object indexes the starting operation of the sequence with the "!entryPoint" key (line 14), and stores another JSON object entry per operation in the sequence (line 15). This object contains an entry called "cacheCode" (line 16) that indicates from which previous operations a parameter is needed from the result cache, and the "nextOps" key contains an array of operation names that can follow this operation (lines 17-21).

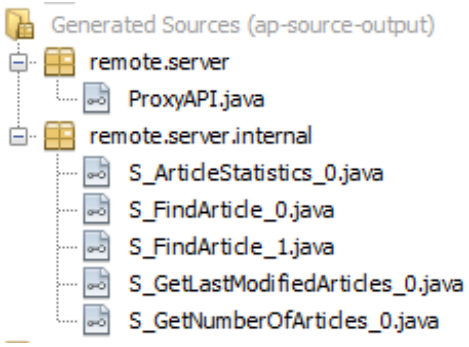


FIGURE 6.2: S-DiSACA layered interface generation example.

It is from this metadata that the Business Schemas interfaces are generated and implemented. However, the interface generation process cannot just create a single Business Schema interface with all the received operations, as one of the objectives is to ease the development burden of not having to master the defined sequences. Thus, the Business Schemas are generated in layers, as shown in

Fig. 6.2 shows an example implementation of a set of Business Schemas from a very simple set of flowcharts. Two types of interfaces are generated, a *ProxyAPI* interface meant to be instantiated by the client application and a set of flowchart Business Schema interfaces.

The flowchart Business Schemas follow a naming convention of the type "S_<flowchartName> _<id>". The "S" indicates that this class refers to a sequence of operations, and distinguishes this type of Business Schemas from those implemented previously. "<flowchartName>" states which flowchart the interface belongs to and finally "<id>" uniquely identifies the flowchart. Thus, "S_FindArticle_0", which can be instantiated from the ProxyAPI, is the root (id 0) Business Schema of the "FindArticle" flowchart. After executing an operation in this Business Schema, the "S_FindArticle_1" Business Schema is automatically instantiated which contains the next available operations according to the flowchart.

Fig. 6.3 shows an example interface declaration of the ProxyAPI, and it contains a function for each flowchart that the client application is authorized to execute. These functions instantiate the Business Schema associated with the root node in each respective flowchart, which allows executing the associated operation. The functions have the naming convention "<flowchartName>_<operationName>()", so "FindArticle_searchArticles()" instantiates a Business Schema that allows to execute the "searchArticles" operation related to the "FindArticles" flowchart.

Lastly, the final step during the compilation process of the client application is to generate the Business Schema interfaces, an example of which is shown in Fig. 6.4.

The Business Schema interfaces have two types of functions declared: a single "execute" function, which executes the operation associated with this Business Schema; and a set of other functions whose name corresponds to other operations in the flowchart and that instantiates the next respective Business Schema. The execute function can be implemented with any number of parameters, in this case, the "searchArticles" operation requires a "query" parameter to find related articles for. The "consumer" parameter is always present and allows the client to process the reply. The "execute"

```
1   package remote.server;
2
3   import remote.server.internal.*;
4
5   public interface ProxyAPI {
6
7       public S_GetLastModifiedArticles_0
8           GetLastModifiedArticles_getLastModifiedArticles();
9       public S_ArticleStatistics_0
10          ArticleStatistics_getArticleModificationCount();
11      public S_FindArticle_0
12          FindArticle_searchArticles();
13      public S_GetNumberOfArticles_0
14          GetNumberOfArticles_getNumArticles();
15  }
```

FIGURE 6.3: S-DiSACA proxy interface example.

```
1   package remote.server.internal;
2
3   import java.util.function.Consumer;
4   import sdraca.common.model.APIResult;
5
6   public interface S_FindArticle_0 {
7
8       public S_FindArticle_0 execute(String query,
9               Consumer<APIResult> consumer);
10      public S_FindArticle_1 getArticleByTitle();
11  }
```

FIGURE 6.4: S-DiSACA Business Schema interface example.

function also returns a reference to the same Business Schema, which allows the client to chain function calls. All other functions used to instantiate the next Business Schemas take no parameters.

After all Business Schema interfaces are generated, the client application can finish its compilation process and any changes to the access control policies are reflected in the Business Schemas available.

6.1.2 Business Schema Interface Implementation

After the Business Schema interfaces have been generated and the client application developed using them, the client application is ready to be used. The client application begins by configuring the Business Manager, which requests once again the most updated metadata from the Policy Manager (after connecting and authenticating) and proceed to implement the Business Schemas.

Fig. 6.5 shows the implementation of the "S_FindArticle_0" interface. The ProxyAPI is not shown but its functions are similar to the "getArticlebyTitle()" function, where a Business Schema factory is used to instantiate a particular Business Schema and validates that the flowchart is indeed being traversed correctly. The Business Schemas also have access to the Sequence Controller (line 11) and a

```
1   package remote.server.internal;
2
3   import java.util.function.Consumer;
4   import sdraca.common.model.APIResult;
5   import sdraca.common.security.sequence.SequenceController;
6   import sdraca.manager.ISequenceFactory;
7   import sdraca.manager.IRemoteExecutor;
8
9   public class S_FindArticle_0Impl implements S_FindArticle_0 {
10
11      private final SequenceController controller;
12      private final IRemoteExecutor executor;
13      private final ISequenceFactory factory;
14
15      public S_FindArticle_0Impl(SequenceController controller, IRemoteExecutor executor,
16              ISequenceFactory factory) {
17          this.controller = controller;
18          this.executor = executor;
19          this.factory = factory;
20      }
21
22      public S_FindArticle_0 execute( String query, Consumer<APIResult> consumer) {
23          consumer.accept(executor.remoteExecute("FindArticle", "searchArticles", query));
24          return this;
25      }
26
27      public S_FindArticle_1 getArticleByTitle() {
28          return factory.instantiateSequenceClass(S_FindArticle_1.class);
29      }
30  }
```

FIGURE 6.5: S-DiSACA Business Schema implementation example.

Remote Executor that sends the "Execute" command to the Policy Manager and handles the remote execution process.

Thus, the "execute" function simply invokes the "remoteExecute()" function in the Remote Executor, passing the flowchart name, operation name, and any parameters it receives. The result of this call is then sent to the consumer so that the client application can easily process it (line 23). Fig. 6.6 shows a slightly different implementation of the "execute()" function, which belongs to the "S_FindArticle_1" Business Schema. This Business Schema is associated with the "getArticleByTitle" operation, and as such one would expect it to receive as a parameter the "title" of the article to search for. However, since a previous Business Schema is used to obtain a list of articles titles, the "title" parameters can come from the result cache.

This process is shown in this figure, where the Remote Executor is still used to execute the underlying operation, but the "title" parameter is obtained from the result cache within the Sequence Controller (line 24). The relevant sequence data is obtained and the cached result from the "searchArticles" operation is used. Note that the operation names from where to obtain the required parameters are sent in the "cacheCode" entry of the metadata associated with each operation (Listing 6.1).

With this implementation process completed, the Business Manager is then able to compile the

```
23      public S_FindArticle_1 execute(Consumer<APIResult> consumer) {
24          consumer.accept(executor.remoteExecute("FindArticle", "getArticleByTitle",
25                  controller.getSequence("FindArticle").getCachedResult("searchArticles")));
26          return this;
        }
```

FIGURE 6.6: S-DiSACA Business Schema implementation example with result cache.

generated classes and to instantiate each of them as necessary, allowing the client to issue the execution of operations over the data until termination.

6.1.3 Business Schema Usage

To show how all of these classes and interfaces are put together and used, an example client application is shown in Fig. 6.7.

The figure shows the main class, named "Example", that is annotated with a custom "DACAManagedApplication" annotation (lines 20-23). This annotation is the compilation tool that manages the generation of the Business Schema interfaces every-time the application is compiled (note that even though the password is hard-coded for display purposes, in a real application it would be requested to the user).

Looking at the main function, the first important piece of code is in lines 32 and 33, where the Business Manager is configured by passing the authentication credentials and the IP address of the authentication service. This prompts the implementation of the generated interfaces, and any mismatch between them results in an exception at this stage. After this process is complete, the application is now able to access and manipulate the data.

First, the client application should instantiate the ProxyAPI discussed previously using a special "instantiateProxyAPI()" function in the Business Manager (line 35), which uses a class loader to instantiate the interface from its implementation. Then, the client application can select the "searchArticles" operation from the "FindArticles" flowchart and execute it with the provided article name (line 38). Note that since the ProxyAPI contains only the authorized flowcharts, a developer can easily identify which use cases are available and what the first operation is expected to be. The client application then provides a consumer, which in this case simply filters the results to find an article name identical to the one desired (lines 39-40). If such a title name is found, then it is set as a parameter in the reply (line 41).

After this is done, the client application requests the next Business Schema in the flowchart ("getArticleByTitle", line 42) and executes it passing another consumer that prints the article and the error message received (lines 43-45). Note that the interface allows only to get operations that follow the current flowchart, meaning that a developer can easily follow a flowchart without having to master any of them. Additionally, the client application does not set the article title when executing, as the desired title was already set in the previous step as a protected parameter.

The client application then executes a different flowchart, requesting the 5 most recent modified articles (lines 49-51), and terminates.

```java
@DACAManagedApplication(
        username = "regateiro",
        password = "secretpwd"
)
public class Example {

    private static final String ARTICLE_NAME = "Aveiro, Portugal";

    public static void main(String[] args) throws Exception {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        //Configure Environment and Load BusinessSchemas
        try (IBusinessManager manager = BusinessManager.configure(
                "regateiro", "secretpwd", "localhost", 9000)) {

            ProxyAPI API = manager.instantiateProxyAPI(ProxyAPI.class);

            System.out.println(String.format("Articles related to %s:", ARTICLE_NAME));
            API.FindArticle_searchArticles().execute(ARTICLE_NAME, (reply) -> {
                reply.getResults().stream().filter((result)
                        -> new JSONObject(result).getString("_id").equalsIgnoreCase(ARTICLE_NAME)
                ).forEach((result) -> reply.setAsParameter(result));
            }).getArticleByTitle().execute((reply) -> {
                System.out.println("\nArticle with the exact name:\n"
                        + new JSONObject(reply.getFirstResult()).toString(3));
                System.out.println("Error Message: " + reply.getErrorMessage());
            });

            System.out.println("\nLast 5 modified articles:");
            API.GetLastModifiedArticles_getLastModifiedArticles().execute(5, (reply)
                    -> reply.getResults().stream().forEach((result) -> System.out.println(result))
            );

            System.out.print("\nPress enter to terminate...");
            in.readLine();
        } catch (IllegalStateException ex) {
            System.out.println("Somewhere the method has been denied execution.");
            System.out.println("Details: " + ex.getMessage());
        }
    }
```

FIGURE 6.7: S-DiSACA client example.

6.1.4 Operation Sequencing

Similarly to the parameter protection feature, the operation sequence in the S-DRACA was only applied on the client application for ease of use. It allowed the application developers to quickly follow the defined use cases without mistakes. However, it could be easily bypassed by using reflection mechanisms to disable the sequence controller. Thus, in the S-DiSACA a sequence controller was also placed on the Policy Manager to ensure that the data access followed the predefined sequences of operations. This sequence validation process is performed during the validate call in Fig. 6.9 and is more clearly detailed in Fig. 6.8.

The diagram shown in this figure shows most of the process required to execute an operation. First, the Business Manager in a client application connects to the Policy Manager worker responsible to handle its requests, which performs some credential validation (explained in the next section).

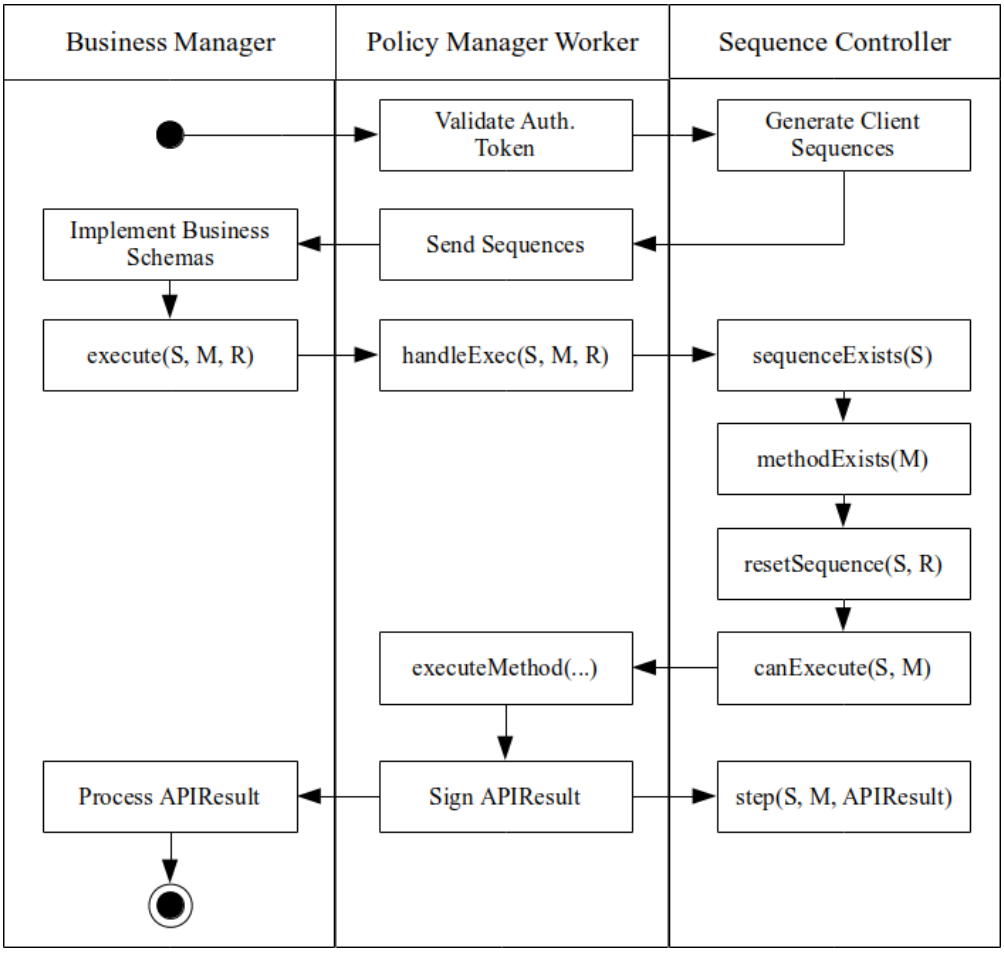


FIGURE 6.8: S-DiSACA remote execution diagram with operation sequencing.

After the client is authenticated, the Policy Manager instantiates a sequence controller for the client, generating in the process the operation sequences that the client application is authorized to execute. These sequences are stored as graphs that the sequence controller can easily navigate.

From these sequences, the Business Manager can implement the various Business Schemas that the client application was written to use. These are then used by the application to execute various operations, sending the "Execute" command along with the sequence name S being executed, the method name M (i.e. the action in a flowchart context) and a flag indicating if a new sequence is being initiated (R). The Policy Manager then handles the request by checking with the sequence controller if the sequence and the method both exist, resets the sequence if the flag R indicates that the client is starting a new sequence (resetting the user context) and if the method indicated is in one of the nodes that follow the current node in the sequence. If everything checks out, the Policy Manager then requests the execution of the method to the API handler and obtains an APIResult object.

Before sending the APIResult object to the client application, the Policy Manager signs it, a crucial step in the communication security that is explained in section 4.3. The APIResult is then sent both to

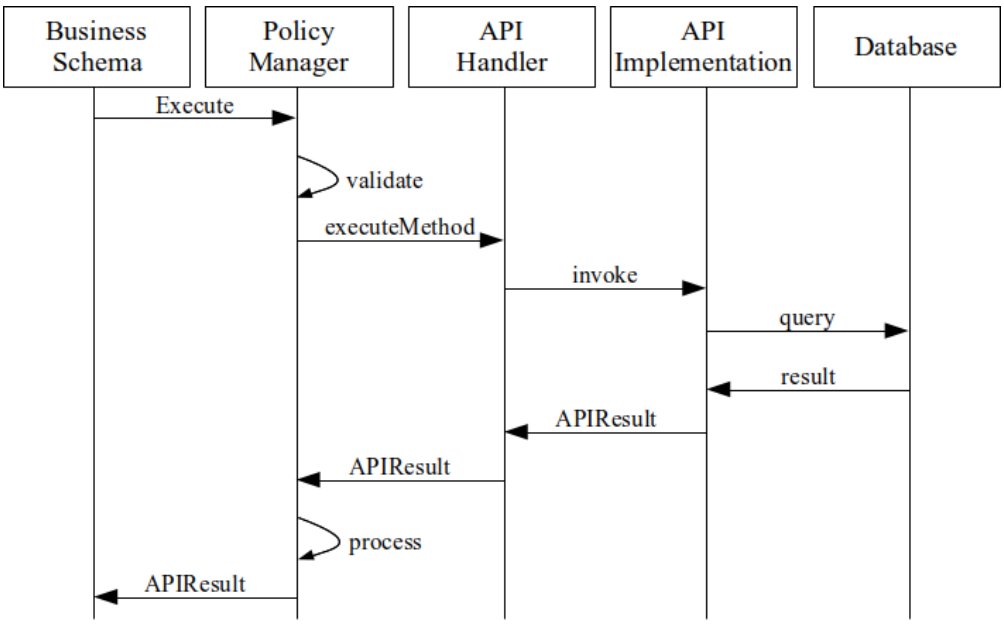


FIGURE 6.9: S-DiSACA remote execution call flow.

the client application for processing and the sequence controller to cache the valid results that can be used in posterior executions as parameters.

The client application also performs these checks locally to prevent any attempt to execute methods that violate the defined sequences, and the Business Schemas are implemented in such a way that the provided interface prevents methods from being executed out of order. This aims to ease the development burden that such an approach would bring.

6.1.5 Remote Execution

In this section, the implementation approach regarding the remote call execution of operations through the Business Schemas along with the protection of the operation queries and their parameters is detailed.

The call flow used in the S-DiSACA to remotely execute a database access method is shown in Fig. 6.9. It begins at the point where the client application executes a method in one of the Business Schemas. This method sends an "Execute" command to the Policy Manager requesting the execution of the operation with the same name. The Policy Manager then validates the request, ensuring that the method name exists and that the number of parameters matches, along with several other security checks that are detailed in the next section. If every security check is satisfied, the Policy Manager requests the execution of the method to the API Handler, which can invoke said method via reflection. The invoked API implementation method then performs its logic to query the underlying database.

Once the API implementation receives the result, the API Handler constructs an APIResult object, which the Policy Manager then processes and sends back to the client application. The APIResult

processing, among other things, saves the result in its result cache so that it can validate future execution calls if necessary.

The Business Schema also processes the result to cache it to validate future execution calls of other Business Schemas when necessary. Note that this process can be manipulated by malicious users as previously mentioned. However, since the Policy Manager now also verifies each execution and the parameters used, such an attempt results in a runtime execution error contained in the received APIResult object.

6.1.6 Communication Security and Data Integrity

The communication security in the S-DiSACA is concerned mostly with the confidentiality and integrity of the data used within the architecture. These are broad terms that can be applied in multiple layers of the architecture, some already discussed.

However, the question of how an encrypted communication channel can be created that authenticates both the server and the client application as legitimate entities yet stands, as well as how to guarantee that the data returned by the database is used accurately within the architecture for parameter protection. Both of these concepts are analysed and discussed in this section.

Data Integrity

The process by which the S-DiSACA can remotely execute operations has been detailed, however the way that the results obtained from the database are stored and their integrity guaranteed has yet to be discussed.

As previously mentioned, the database results are transmitted through the network as APIResult objects. Fig. 6.10 shows the methods and fields used by this class.

Most of these methods and fields are straightforward, they are used to store the *operation* that generated the *results*, the results to be used as *parameters* in a follow up operation and the eventual *error* message. However, a malicious user may be able to attack these objects and modify any of these fields to achieve various results.

Fortunately, the operation and the parameters are already protected as discussed earlier in this chapter. However, it could still try to alter the sequence identifier *sequenceUUID* in an attempt to manipulate the available parameters and the operation to execute next by using an APIResult from one flowchart in another. Since the same application can try to execute different flowcharts at the same time, this would be possible.

To prevent this type of attack, the APIResult objects are created with the safekeeping of the data integrity in mind. To achieve this goal, the APIResult objects must be signed by the Policy Manager. This *signature* exists in the object itself and it depends on the *operation*, *results*, *error* message, and *sequenceUUID* fields. This field is set using the *sign* method, which also sets the Policy Manager sequence identifier. Then, anyone with the Policy Manager public key can use the *verifySignature* method to verify the signature on the object and to ensure that none of the related fields have been modified.

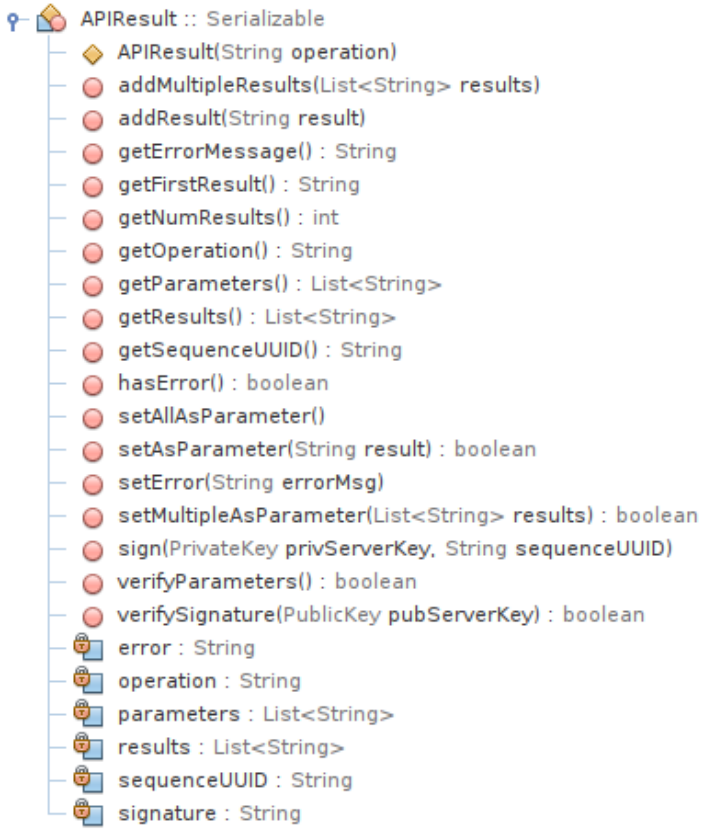


FIGURE 6.10: APIResult class methods and fields.

Note that the *parameters* field is not verified with the signature, this is because the results to be used as parameters are set by the client application, not the Policy Manager. However, the client application cannot modify the sequence identifier without the Policy Manager detecting, and the parameter protection feature of the S-DiSACA ensures that the parameters set are always valid.

Encrypted Communication

In this section, the implementation of encrypted communication via Secure Sockets Layer (SSL)/Transport Layer Security (TLS) with a pre-shared key is further detailed.

To reiterate, the approach discussed in section 4.3.2 introduced the idea of modifying the SSL/TLS master key S with a pre-shared key PSK known only to the legitimate client and server. This means that once this process completes and the communication remains intelligible then both parties are authenticated.

Eq. 4.17 has shown the process by which the master key S can be modified. Therefore, it is only required to call the "ChangeCipherSpec" procedure on the SSL/TLS socket used for communication to ensure that the read and write cyphers are updated. Fig. 6.11 and 6.12 show how this process can be achieved at a high level in a server and a client application, respectively.

```
32    SSLServerSocket serverSocket = (SSLServerSocket) ssf.createServerSocket(port);
33    serverSocket.setEnabledCipherSuites(new String[]{"TLS_DH_anon_WITH_AES_128_CBC_SHA"});
34    SSLSocket clientSocket = (SSLSocket) serverSocket.accept();
35    Object handshaker = ReflectionUtils.getFieldValue(clientSocket, "handshaker");
36    clientSocket.startHandshake();
37    changeSSLConnectionKeys(clientSocket, handshaker, secret);
38    return clientSocket;
```

FIGURE 6.11: Java server-side TLS pre-shared key application[38].

```
46    SSLSocket socket = (SSLSocket) csf.createSocket(dest, port);
47    socket.setEnabledCipherSuites(new String[]{"TLS_DH_anon_WITH_AES_128_CBC_SHA"});
48    Object handshaker = ReflectionUtils.getFieldValue(socket, "handshaker");
49    socket.startHandshake();
50    changeSSLConnectionKeys(socket, handshaker, secret);
51    return socket;
```

FIGURE 6.12: Java client-side TLS pre-shared key application[38].

Once a socket is created, a cypher suite of the type "TLS_DH_anon..." is used. This cypher states that the key exchange protocol used is Diffie-Hellman, but more importantly that no authentication is performed (anon), which means that certificates are no longer required to establish the connection. Then, the object responsible for setting the connection key is obtained from the socket via reflection (the handshaker object in the example) and accessed to manually modify the key agreed between the client and the server.

Fig. 6.13 shows the process of changing the TLS key for the Oracle Java 8 implementation of the TLS protocol. While this process is different from other implementations, the underlying idea remains the same. For reference, reading or modifying a field through reflection requires 3 steps: first, a reference to the field or method of a class must be obtained (e.g. line 85); second, the field or method must be set as accessible (line 86); and third, the reference can be used to get, set, or invoke the respective instance on an object of the associated class (line 87).

Lines 85 to 87 save the connection state of the socket, which is needed later. Then, the *masterSecret* is obtained from the socket (line 90-93). This *masterSecret* is our key S and as such it is modified using the pre-shared secret *secret* (lines 96-99).

Then, a reference to the *changeConnectionKeys* method is obtained and invoked with the new key S (lines 107-110). This creates two separate internal keys, one for reading and another for writing data. Once these two internal keys are generated, they must be used to create the read and write cyphers. This process requires the handshaker to be set with the state *cs_HANDSHAKE* (lines 113-118). Lastly, the references to the *changeReadCiphers* and *changeWriteCiphers* are obtained and invoked so that the TLS socket effectively uses the new keys (lines 121-128) and the previous connection state is reset (line 131).

In the example above for Oracle Java 8, the *secret* had already been transformed and hashed with a salt value, so $secret = H(F(PSK))$, and the process that modifies the *masterSecret* (S) just hashes both of these values together, changing S into S'.

```
84  // Retrieve the previous connection state
85  Field connStateField = socket.getClass().getDeclaredField("connectionState");
86  connStateField.setAccessible(true);
87  int prevState = connStateField.getInt(socket);
88
89  // Retrieve the master secret
90  Field masterSecretField = socket.getSession().getClass().getDeclaredField("masterSecret");
91  masterSecretField.setAccessible(true);
92
93  SecretKey masterSecret = (SecretKey) masterSecretField.get(socket.getSession());
94
95  // Hash the current master secret with the shared secret
96  MessageDigest md = MessageDigest.getInstance("SHA-512", "SUN");
97  md.update(masterSecret.getEncoded());
98  md.update(secret);
99  byte[] newMasterSecretBytes = Arrays.copyOfRange(md.digest(), 0, masterSecret.getEncoded().length);
100
101 // Set the newly created secret in the master secret
102 Field keyField = masterSecret.getClass().getDeclaredField("key");
103 keyField.setAccessible(true);
104 keyField.set(masterSecret, newMasterSecretBytes);
105
106 // Invoke the method to calculate the connection keys
107 Method calcConnKeysMethod = handshaker.getClass().getSuperclass()
108         .getDeclaredMethod("calculateConnectionKeys", SecretKey.class);
109 calcConnKeysMethod.setAccessible(true);
110 calcConnKeysMethod.invoke(handshaker, masterSecret);
111
112 // Set the handshaker with the new connections keys in the SSL Socket
113 Field handshakerField = socket.getClass().getDeclaredField("handshaker");
114 handshakerField.setAccessible(true);
115 handshakerField.set(socket, handshaker);
116
117 // Set the current socket state to cs_HANDSHAKE
118 connStateField.set(socket, ReflectionUtils.getFieldInt(socket, "cs_HANDSHAKE"));
119
120 // Invoke the method to change the read cipher
121 Method changeReadCiphersMethod = socket.getClass().getDeclaredMethod("changeReadCiphers");
122 changeReadCiphersMethod.setAccessible(true);
123 changeReadCiphersMethod.invoke(socket);
124
125 // Invoke the method to change the write cipher
126 Method changeWriteCiphersMethod = socket.getClass().getDeclaredMethod("changeWriteCiphers");
127 changeWriteCiphersMethod.setAccessible(true);
128 changeWriteCiphersMethod.invoke(socket);
129
130 // Put the socket back to its original state and set the handshaker to null
131 connStateField.set(socket, prevState);
```

FIGURE 6.13: TLS key modification procedure for Oracle Java 8.

This approach was presented in [38]. However, since it uses reflection mechanisms to modify the TLS keys, it is dependent on the TLS implementation and may not work once the development kit for a programming language is updated. Thus, a fallback method is also presented in Fig. 6.14, which uses a public key from the server to encrypt a randomly generated secret s. This secret is sent by the client to the server to use as the encryption key for the communication going forward.

However, this fallback approach does not authenticate the client and malicious users may be capable of impersonating the servers by installing their own certificate authority on the client applications. To authenticate both of them, a simple challenge-response mechanism is used to ensure that both

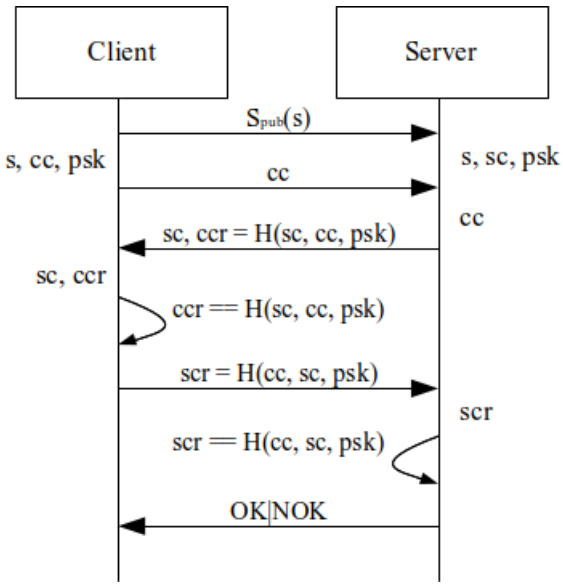


FIGURE 6.14: Mutual challenge-response authentication protocol.

parties know the pre-shared secret.

After the communication is encrypted, this mutual authentication takes place. To the left and right of the vertical lines is shown what pieces of information the client and the server owns, respectively, and when they acquire it. First, the client generates the client challenge *cc* and the server challenge *sc* randomly. The client sends its challenge *cc* to the server, which it uses to calculate the client challenge-response *ccr*. It then sends the *sc* and the *ccr* to the client. The client calculates the expected client challenge-response and compares it to the received value. If they are the same, then the server has proven to know the pre-shared key and the process continues, otherwise, the connection is dropped. Then, the client calculates the server challenge-response *scr* from the *sc* and sends it to the server, which also calculates the expected value and compares it to the one received from the client. Once again, if they are the same then the authentication succeeded, otherwise the connection is dropped.

If the challenge-response exchange succeeds, then data may be exchanged in the encrypted communication channel. The major downside of the fallback method to the TLS with pre-shared key proposed above is that it does not follow a standard, and could be vulnerable to more obscure attacks.

This process has also been secured against one common attack known as a reflection attack. This attack has a malicious user connect to the server until he receives the server challenge *sc*. Then, a secondary connection is established where the client sends the challenge *sc* back to the server to obtain the correct response *scr*. That response is then sent to the server on the first connection to authenticate successfully. To prevent this attack, the *cc* and the *sc* challenges are modified so they always end in 0 and 1, respectively. This way, during the second connection in the above attack scenario, the server can detect that it was sent a server challenge instead of a client challenge and drop the connection immediately. Finally, Man-In-The-Middle attacks are also prevented by the initial encrypted secret *s* that only the legitimate server can decrypt.

Security Considerations

It is important to consider the implications that these two approaches have on the security of the communication. In [111], three security considerations are presented that should be taken into account when dealing with communication protocols such as SSL/TLS.

The first consideration is regarding perfect forward secrecy [116] and it expresses the property of a communication protocol to not compromise past session keys if the long-term keys are compromised. Considering that the adapted protocol uses the Diffie-Hellman private key exchange, which generates a different key S for each handshake of the protocol, when the key S' is compromised it is only possible to decrypt that communication session, given that the malicious user is also in possession of the client pre-shared key. Since the agreed keys S are independent of one another, all past communications remain uncompromised, providing perfect forward secrecy. The same applies to the fallback method since the key generated by the client that is encrypted using the servers public key is randomly generated for every new communication channel created and they are all independent of one another.

The second consideration regards to brute-force and dictionary attacks. The use of a fixed shared secret of limited entropy such as a pre-shared key chosen by a human (e.g. a password) may allow a malicious user to perform a brute-force or dictionary attack to recover the shared secret. This may be executed as an off-line attack (against a captured TLS handshake message) or as an on-line attack where the malicious user attempts to connect to the server and tries different keys. In the case of a protocol that uses Diffie-Hellman, such as the first method proposed, the malicious user can only obtain the message it requires by getting a valid client to connect to him, for example by using a Man-In-The-Middle attack. The same can happen in the fallback method when the malicious user impersonates the server by installing his own certificate authority on the client application. While a weak pre-shared key can be obtained from such methods, only future communications between the client and the server are vulnerable, since the key S changes every time a new connection is established in both methods.

Additionally, since a Man-In-The-Middle attack is required to obtain the data needed for an offline attack, it is always detectable because the malicious user won't be able to replay the communication before carrying out the attack. Given that the authentication fails for that first Man-In-The-Middle attack, if the system triggers a forced pre-shared key reset then future communications are not vulnerable. However, as with many other protocols, a malicious user could carry out a denial of service attack by making Man-In-The-Middle attacks on every communication attempt.

Finally, considering identity confidentiality, currently both approaches send the client identity encrypted with the server public key. However, since the client identity is required for the server to know which pre-shared key to use to modify the agreed key S, this method would only prevent eavesdroppers from knowing the communicating parties. In a Man-In-The-Middle attack scenario, the malicious user would still be able to know the identity of the client.

6.2 Configuration and Usability Costs

One critical aspect that has not been addressed yet is the usability cost of using an architecture such as the one presented here. Every feature that aims to improve the security of a system always brings some sort of drawback, such as overhead on computational resources (i.e. CPU time, memory, storage, etc.) and/or usability complexity in the form of additional configuration steps and prompts.

When it comes to the usability complexity, increasing the security requirements may lead to users struggling with using the architecture [108], which in turn makes them choose more convenient and functional solutions. Alongside security, usability played a key role in the design of the architecture that lead to the careful design of the interface of the generated Business Schemas and ProxyAPI. Furthermore, the configuration of the policies to be used in the Binary Decision Fuzzy Inference System (BDFIS) to grant permissions to each user was also made easier by a couple of simple tools.

The usage of the ProxyAPI and the Business Schemas was already covered in section 6.1, including how they are generated, implemented and used. Section 6.1.3 also demonstrates how these interfaces allow a developer to follow the different flowcharts without having to master any of them.

The main outstanding issue that remains is the cost of configuring this architecture to be used by developers, which includes the following steps:

1. Create the access control policies;
2. Implement a server-side data access API;
3. Define the operations used in the data access API and the flowcharts that use them;
4. Define the permissions and associated flowcharts;
5. Create a record for each new user/application.

These steps naturally incur a cost to be carried out when compared to simpler solutions already available. However, these are one time costs that will result in lower application development, debugging, and user management times as most of the policy-aware data access code generation is automated. Furthermore, the generated interfaces allow developers to easily follow the defined flowcharts, which naturally satisfies the access control policies. Lastly, if more than one application needs to be developed that accesses the same data, every common flowchart and operation can be reused to automatically generate the client-side data access layer.

While access control policies can be written using standard Fuzzy Control Language (FCL) and the server-side data access API implemented using an IDE, the relations between the access control policies, their permissions, the flowcharts and the data access API operations still need to be defined. In this proof-of-concept, a simple tool was used to insert and manage these relations in an SQLite database, a file-based relational database. The architecture can read the relations from this database to initialize the flowcharts internally and then associate the authorized flowcharts with the users. While a similar tool was also used to manage the users, they should be automatically registered in a real world scenario to avoid human intervention. These tools are first described in this section, and possible better alternatives discussed afterwards.

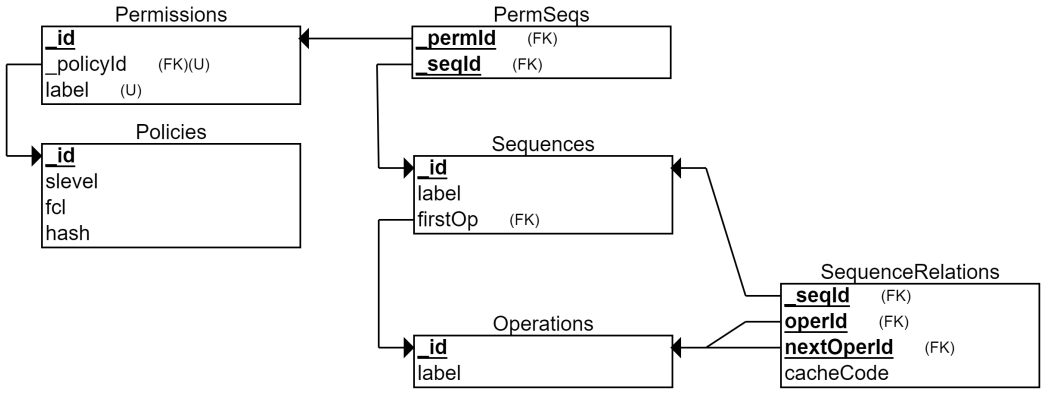


FIGURE 6.15: PolicyDB relational schema.

6.2.1 Configuration Tools

Since there are tools available to parse and validate FCL files and an Integrated Development Environment (IDE) can be used to write the data access APIs, what remains is to create the user/application records and the mapping between the policies, permissions, flowcharts and operations.

Fig. 6.15 shows the database schema used to map the written policies with the permissions, flowcharts and operations. First, a policy must be written in FCL and then the Java tool can insert it into the policy database. A policy must be associated with a security level, a label used to describe the policies applicable to the data being protected. An hash of the FCL policy is also stored alongside it to allow the architecture to quickly detect changes in policies. The permissions that a particular FCL policy makes access control decisions for are also stored in the *Permissions* table.

The flowcharts are stored in the *Sequences* and *SequenceRelations* tables. The former stores the flowchart name and the first operation. The latter stores the edges of the flowchart by associating an operation with the next allowed operation. The cacheCode is used by the architecture to retrieve a parameter from a previous operation with the specific label from the results cache. Finally, the *Operations* table stores every operation available in the data access API and the *PermSeqs* table relates permissions with the flowcharts they authorize. The Java tool requests all the necessary information and stores it in this database for the architecture to use.

Fig. 6.16 shows the available commands on the policy management tool. It allows to list, add and remove operations, sequences and policies. When adding sequences or policies, the operations that they use or reference must have already been defined. For sequences, the tool requests the sequence name and then the first operation. Afterwards, it will display the first operation and request the operations allowed to execute after it, repeating for each new operation referenced. Finally, when adding policies it will request the security level and a path to the FCL file, which is parsed and validated before being stored. The permissions handled by each policy are also requested at this step.

Regarding the user and authentication data, Fig. 6.17 shows the database schemas of the databases resposible for storing them. The authentication database is used by the authentication service, which

```
Connecting to database /home/regateiro/Document
Checking tables...
Ready!

*** Available Commands ***
listops         Lists the existing operations.
addops          Adds api operations.
delops          Deletes api operations.
listseqs        Lists the existing sequences.
addseq          Adds a sequence.
delseq          Deletes a sequence.
listpols        Lists the existing policies.
addpol          Adds a policy.
delpol          Deletes a policy.
help            Displays this information.
exit            Close this application.

>
```

FIGURE 6.16: Policy management tool interface.

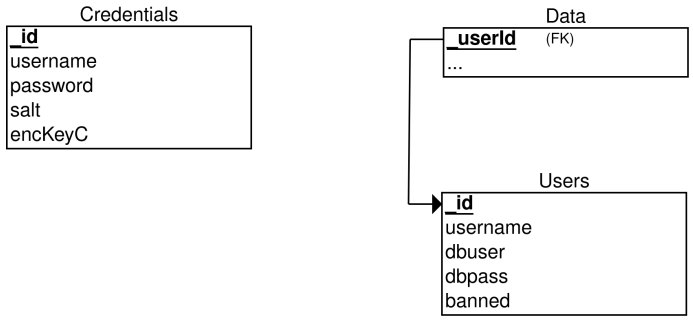


FIGURE 6.17: AuthDB (left) and UserDB (right) relational schemas.

is meant to store the user credentials and the user encoded key *C* mentioned back in section 4.3. The user database stores the database credentials for each user in the table *Users* and the data used to feed to BDFIS to determine which permissions are granted in the table *Data*. The latter can be obtained from a trusted third-party instead if available.

Fig. 6.18 shows the commands available in the tool that manages the users. It simply allows to register a new user, which requests the username, password and other user specific data required by the BDFIS for the access control decision making process. The database credentials are automatically generated and stored encrypted by the symmetric key *C* as previously discussed.

```
Connecting to database /home/regateiro/Docu
Checking tables...
 > Creating table Users... OK!
OK!

Ready!

*** Available Commands ***
register        Register a new user.
help            Displays this information.
exit            Close this application.

>
```

FIGURE 6.18: User management tool interface.

6.2.2 Alternative Configuration Methods

The development of these tools were not the main focus of this thesis, and as such only a basic solution was developed to showcase the proof-of-concept. However, using the tools shown above in a real-world scenario poses some problems.

It is not always the case where a database application is developed from scratch. In many cases, a codebase already exists that accesses the database directly and it already adheres to the access control policies if it was validated, albeit implicitly. In these scenarios, instead of a tool that allows to define the policies a parser could be used. The parser could analyse the codebase and infer the flowcharts used.

However, a tool to validate the policies both created manually and potentially generated by parser is necessary. Considering that projects can model their requirements using UML diagrams, a tool that maps the designed diagrams with the access control policies, flowcharts and permissions would be essential to increase the trustworthiness of the security experts in the system using the architecture.

While the initial cost of using this architecture is greater than using simple database connectivity tools and solutions, the initial time investment can be reduced with these (and other) more sophisticated tools. Furthermore, it is important to understand that the time is compensated at the later stages of application development, as the data access layer code is mostly automatically generated, correct and robust, allowing developers to quickly and easily follow the established access control requirements and policies. Finally, the soft access control policy support also enables users to be quickly authorized provided a trustworthy source of user data is available, reducing the management load in scenarios with highly desired data.

6.3 Correctness Evaluation

In this section, the correct operation of the S-DiSACA as presented is evaluated. The implementation was made in Java and follows the architecture shown in Fig. 6.1.

TABLE 6.1: Test case flowcharts.

#	Flowchart	Operations
1	FindArticle	searchArticles -> getArticleByTitle
2	BanUser	getUsers -> banUser
3	UnbanUser	getUsers -> unbanUser
4	UpdateArticle	updateArticle
5	GetLastModifiedArticles	getLastModifiedArticles
6	GetNumberOfArticles	getNumArticles
7	CorrectArticle	see Fig. 6.19
8	DeleteArticle	deleteArticle
9	ArticleStatistics	getArticleModificationCount

The client application is built upon the example shown in Fig. 6.7, where each attempt to access the Wikipedia article data stored in a MongoDB instance is modified in some way to test a separate security feature. The MongoDB was chosen to store the Wikipedia articles because it is optimized to store documents and it would also allow testing the architecture with a non-relational data store. The PDP is a BDFIS, built using the jFuzzyLogic library, that is created using policy files written in FCL. The policy server and user data stores were implemented using SQLite.

6.3.1 Test Subject and Policies

While the complete API to access the articles is shown in Fig. 4.1, when using a subject it should only have access to the subset of operations it is authorized to use.

This authorization is determined by three different aspects: the subject's parameters that are provided to the system; the FCL policy files that define how those parameters influence whether a permission is granted or denied; and which flowcharts are associated with each permission.

The subject's parameters can be obtained in a myriad of ways, from self-collection to requesting them to a trusted third-party. This is crucial to ensure that each subject is correctly authorized, however, it is very context-dependent. In this proof of concept, the attributes of the test subject were stored in the user data store manually. In terms of the FCL policy files, three were created for this test scenario: the researcher, business, and administrative policies. Each of these policies also has a *Read* and *Write* permission output and are applied to a security level of the same name. Each of the defined flowcharts was then given a specific security level and permission that would be required to access them. A public security level was also created, but no FCL policy file was defined since any flowchart with this particular security level is meant to be always available to every subject. Finally, nine distinct flowcharts were created that allow a subject to access and manipulate articles and other associated data, shown in Table 6.1.

This table enumerates each defined flowchart and the sequence of operations that defines them. Flowcharts 4, 5, 6 and 8 are single operation sequences, and allow the subjects to execute that operation standalone. Flowcharts 1, 2, and 3 have sequences of two operations, and flowchart 7 is shown in Fig. 6.19 since it is considerably more complex to show in the table. Flowchart 7 allows a subject to

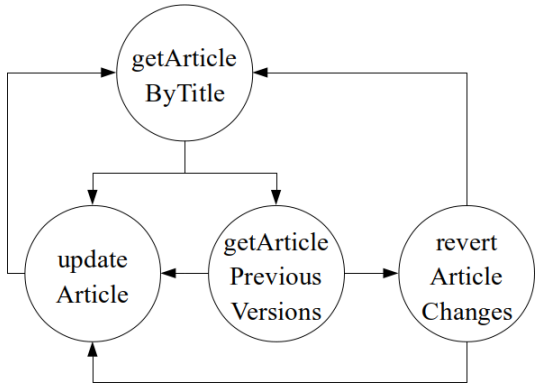


FIGURE 6.19: CorrectArticle flowchart.

TABLE 6.2: Test case flowchart security levels.

#	Flowchart	Permission	Security Levels
1	FindArticle	Read	public
2	BanUser	Write	administrative
3	UnbanUser	Write	administrative
4	UpdateArticle	Write	researcher
5	GetLastModifiedArticles	Read	researcher business
6	GetNumberOfArticles	Read	public
7	CorrectArticle	Write	administrative
8	DeleteArticle	Write	administrative
9	ArticleStatistics	Read	researcher business administrative

correct multiple articles by getting an article, possibly checking its previous versions and updating it or reverting it to one such version.

Table 6.2 shows the defined security levels and which permission is required from each security level to authorize a subject. It is clear from the table that a given flowchart, such as flowchart 5, can have more than one security level associated. This allows subjects to be authorized via different policies, as only one of the policies has to be satisfied. Fig. 6.20 shows the outputs of the BDFIS when the test subject used in this proof of concept connects to the system.

The figure is divided into three columns, once for each policy. The number in parenthesis next to each variable and term indicates the crisp input value in the case of the input variables, the crisp output value in the case of the output variables, and the membership degree/rule strength in the case of the linguistic terms.

The research policy column to the left shows three input variables (*avgDBP*, *NumPublications* and *NumCitations*), two abstract variables (*Activity* and *Expertise*) and two output variables (*Read* and *Write*).

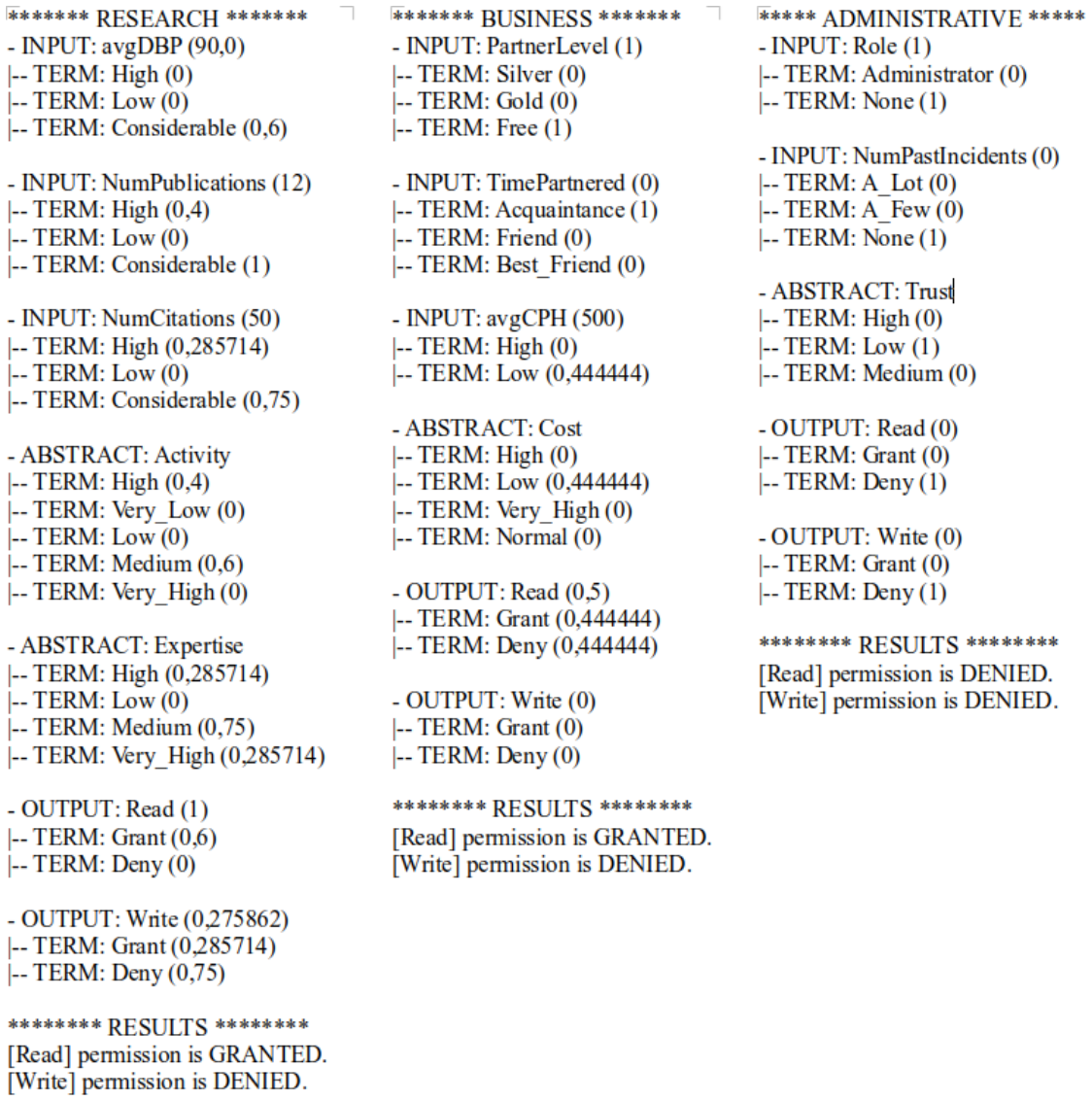


FIGURE 6.20: BDFIS output for the test subject.

It determines the level of expertise of the subject given the number of publications and citations, as well as the degree of recent activity from the average number of days between publications (avgDBP). The activity and expertise levels are then used to determine if the subject is authorized to access the operations protected with the researcher policy via the *Read* and *Write* permissions.

The research policy in the middle column refers to the business policy, which has three input variables (*PartnerLevel*, *TimePartnered* and *avgCPH*), one abstract variable (*Cost*), and two output variables (*Read* and *Write*). It determines how much a partner has cost the company in terms of their partner level, how long they have been a partner and the average number of calls made per hour (avgCPH). The cost level is then used to determine if the subject is authorized to access the operations

protected with the business policy via the *Read* and *Write* permissions.

Finally, the column to the right refers to the administrative policy, which has two input variables (*Role* and *NumPastIncidents*), one abstract variable (*Trust*) and two output variables (*Read* and *Write*). The level of trust is calculated based on the number of past incidents involving the subject, and it is only authorized via the *Read* and *Write* permissions if that subject is both an administrator of the system and the trust level is high enough.

Note that the *Role* input variable linguistic terms, much like the *PartnerLevel* terms in the business policy, are used as Boolean sets. A subject either is an administrator or it is not. This is defined using a singleton function and allows to incorporate both fuzzy and crisp conditions into the decision making process.

The ProxyAPI class shown in Fig. 6.3 was generated using this test subject and allows us to verify that the client application was only provided with the flowcharts that it was authorized to execute. In this scenario, the text subject was granted the *Read* permission in both the research and business policies, which according to Table 6.2 gives access to four flowcharts: FindArticle, GetLastModifiedArticles, GetNumberOfArticles, and ArticleStatistics. These flowcharts are the same ones that were implemented in the ProxyAPI class.

6.3.2 Correctness Scenarios

In this section, several scenarios of the utilization of the S-DiSACA are tested to verify the correctness of the presented features.

The normal usage scenario was shown previously in Fig. 6.7, and it showcased how the client application connects to the S-DiSACA system and issues requests through the available flowcharts.

However, the developers of the client application can attempt to maliciously subvert the defined policies by employing different methods, such as attempting to change the underlying operation to achieve a different outcome. These scenarios are discussed, implemented and tested in the following sections.

Invalid Parameter Insertion

One way that a malicious user can try to subvert the system is by inserting a fake result parameter into the reply received from the Policy Server. With this approach, a malicious user is attempting to have the system execute an operation with an invalid parameter to change the outcome of an operation in some way, such as to potentially corrupt / vandalize the data.

Fig. 6.21 shows how this can be achieved in Java. In line 107, the ProxyAPI object is accessed to issue the execution request for the *searchArticles* operation. When a reply is received, the application extracts the object that holds the results set as parameters from the reply (line 108), adds its own custom parameter into it (line 109). Then, it issues the execution of the following operation in the flowchart (line 111).

However, since the Sequence Controller looks at the result set in the Business Schemas previously executed to find valid parameter values, this approach is immediately thwarted given that the result added as a parameter does not exist in the original result set. The Policy Server replies with the error

```
104 private static void insertInvalidParameterTest(ProxyAPI API) {
105     System.out.println("\n*** INSERT CUSTOM PARAMETER TEST ***");
106
107     API.FindArticle_searchArticles().execute(ARTICLE_NAME, (reply) -> {
108         List<String> parameters = (List<String>) ReflectionUtils.getFieldValue(reply, "parameters");
109         parameters.add("{ my custom parameter }");
110         reply.setAllAsParameter();
111     }).getArticleByTitle().execute((reply) -> {
112         System.out.println("Error Message: " + reply.getErrorMessage());
113     });
114 }
```

FIGURE 6.21: Invalid parameter insertion implementation.

```
92  private static void insertInvalidResultTest(ProxyAPI API) {
93      System.out.println("\n*** INSERT CUSTOM RESULT TEST ***");
94
95      API.FindArticle_searchArticles().execute(ARTICLE_NAME, (reply) -> {
96          List<String> results = (List<String>) ReflectionUtils.getFieldValue(reply, "results");
97          results.add("{ my custom result }");
98          reply.setAllAsParameter();
99      }).getArticleByTitle().execute((reply) -> {
100         System.out.println("Error Message: " + reply.getErrorMessage());
101     });
102 }
```

FIGURE 6.22: Invalid result insertion implementation.

message "One or more APIResult set parameters do not exist as previously obtained results for the current sequence.", indicating precisely that.

Invalid Result Insertion

Another way that a malicious user can similarly try to subvert the system is by inserting a fake result into the result set received from the Policy Server. This is a more carefully thought out approach as the custom parameter is now also set as a result received from the Policy Server. In this way, there is no clear mismatch between the set of result and the set of results to be used as parameters.

Fig. 6.22 shows how this could be potentially achieved in Java. In line 95, the ProxyAPI object is accessed to issue the execution request for the *searchArticles* operation. When a reply is received, the application extracts the object with the results from the reply (line 96) and adds its own custom result (line 97). Then every single result is set as a parameter for the following operation (line 98).

However, upon execution of the next operation (line 99), the Policy Server should still detect the modification of the result set. Since the signature (a signed hash) described in section 4.3 for data integrity no longer matches, the execution is prevented a reply with an error message is returned instead. In fact, upon executing this function the Policy Server replies with the error message "Invalid signature in one or more APIResult parameter.", as expected.

```
116 private static void modifySignatureTest(ProxyAPI API) {
117     System.out.println("\n*** MODIFY SIGNATURE TEST ***");
118
119     API.FindArticle_searchArticles().execute(ARTICLE_NAME, (reply) -> {
120         ReflectionUtils.setFieldValue(reply, "mysignature", "signature");
121         reply.setAllAsParameter();
122     }).getArticleByTitle().execute((reply) -> {
123         System.out.println("Error Message: " + reply.getErrorMessage());
124     });
125 }
```

FIGURE 6.23: Signature modification implementation.

```
127 private static void changeOperationTest(ProxyAPI API) {
128     System.out.println("\n*** CHANGE OPERATION TEST ***");
129
130     API.FindArticle_searchArticles().execute(ARTICLE_NAME, (reply) -> {
131         ReflectionUtils.setFieldValue(reply, "getArticleByTitle", "unbanUser");
132         reply.setAllAsParameter();
133     }).getArticleByTitle().execute((reply) -> {
134         System.out.println("Error Message: " + reply.getErrorMessage());
135     });
136 }
```

FIGURE 6.24: Operation change implementation.

Signature Modification

Since adding a custom result and setting it as a parameter did not work, one might think to change the signature itself. Since this signature is created using a private key that only the Policy Server has access to, creating a new, valid signature for the modified APIResult object is extremely unlikely.

Nevertheless, Fig. 6.23 shows how the signature in the reply object could be modified and tested to verify the correctness of the system. Once again, in line 119, the ProxyAPI object is accessed to issue the execution request for the *searchArticles* operation. When a reply is received, the application modifies the signature in the reply (line 120) to its own custom signature.

Unsurprisingly, the Policy Server replies with the error message "Invalid signature in one or more APIResult parameter.", as the custom signature was no longer created using the Policy Server private key. If the private key is stolen, however, valid signatures could potentially be created. Therefore, the importance of keeping these private keys secure is paramount.

Operation Change

A different approach that a malicious user could attempt to do to subvert the Sequence Controller would be to modify the APIResult underlying operation. While it is true that the operation being modified has already been executed, the operation associated with an APIResult is what is used by the Sequence Controller to determine from which result set a parameter is obtained. Thus, this approach potentially allows results to be used as parameters that are not valid.

```
138 private static void reuseResultBetweenSequencesTest(ProxyAPI API) {
139     System.out.println("\n*** REUSE RESULT BETWEEN SEQUENCES TEST ***");
140
141     APIResult[] firstResult = new APIResult[1];
142     System.out.println("\n - First Execution");
143     API.FindArticle_searchArticles().execute(ARTICLE_NAME, (reply) -> {
144         reply.setAsParameter(reply.getFirstResult());
145         firstResult[0] = reply;
146     }).getArticleByTitle().execute((reply) -> {
147         System.out.println("Error Message: " + reply.getErrorMessage());
148     });
149
150     System.out.println("\n - Second Execution with Result from Another Sequence");
151     S_FindArticle_1 seqClass = API.FindArticle_searchArticles()
152             .execute(ARTICLE_NAME, (reply) -> {
153                 reply.setAsParameter(firstResult[0].getFirstResult());
154             }).getArticleByTitle();
155
156     SequenceController controller = ReflectionUtils.getFieldValue(
157             seqClass, "controller", SequenceController.class);
158     controller.getSequence("FindArticle").cacheResult(firstResult[0]);
159
160     seqClass.execute((reply) -> {
161         System.out.println("Error Message: " + reply.getErrorMessage());
162     });
163 }
```

FIGURE 6.25: Result reuse between flowcharts implementation.

Fig. 6.24 shows how this approach could be implemented. In line 130, the ProxyAPI object is accessed to issue the execution request for the *searchArticles* operation. When a reply is received, the application sets a different underlying operation that generated the reply (line 131) and executes the next operation in the flowchart (line 133).

Again, the Policy Server replies with the error message "Invalid signature in one or more APIResult parameter.", as the signature also depends on the underlying operation. Since the operation was modified, the signature is no longer valid.

Instead of modifying the operation to try to use parameters from an APIResult that is not meant to be used in another operation, the malicious user could use reflection to bypass the ProxyAPI abstraction and send commands directly to the Policy Server. This would allow him to send any operations in any order, but since the Policy Server instantiates the Sequence Controller to verify that the operations being executed follow one of the defined flowcharts, the exception "SEQUENCE VIOLATION" would be generated and thrown back to the client application.

Result Reuse Between Flowcharts

It is clear that, so far, the signature in the APIResult replies thwart any attempt to modify them and have been still being accepted by the Policy Server. Therefore, a malicious user might attempt to not modify an APIResult reply while still using it to his advantage.

Fig. 6.25 shows an exploit implementation that follows this line of thought. An initial flowchart is initiated and the first operation executed (line 143), getting as a reply a valid APIResult with a parameter value that is in some way interesting to the malicious user. This APIResult is saved (lines 145). Then, a different flowchart is initiated and the first operation executed (line 151). However, instead of using a result from the reply obtained during this second execution, the initial APIResult reply is used instead (line 153). Lines 156 to 158 show the initial reply being saved as a valid result of the second flowchart execution. Since none of the APIResult objects were modified, the signatures are still valid, and therefore the Policy Server should accept the initial result from the first flowchart execution in the second.

However, a security feature was added to prevent this scenario. Each APIResult contains a unique sequence identifier that is set by the Policy Server when it is created. This identifier remains the same for every APIResult created during a flowchart execution, but a new one is generated when a new flowchart is initialized. Therefore, executing this approach results in the Policy Server replying with the message "Unexpected sequence identifier in one or more APIResult parameter.", indicating that a sequence mismatch was found in the identifiers.

While it is not shown in the figure, the malicious user could take the sequence identifier from the second reply and set at in the first one, ensuring that the Policy Server does not find this mismatch. Unfortunately, this process modifies the reply object, causing the signature to no longer be valid once again.

6.4 Performance Assessment

In this section, a performance assessment is carried out to demonstrate primarily the overhead that the S-DiSACA introduces in the data access procedure. However, while this overhead can be measured and analysed, the impact of the usage of this architecture during the development stages of a client application is harder to quantify.

Instead of letting developers access and manipulate the data directly on the data store, they must now use a well-defined interface that contains only the data manipulation operations that the application is authorized to execute. This implies that the development effort of writing correct data manipulation logic has shifted from the application developers to an earlier point in the application development life-cycle. However, once the policy files are created, the developers have at their disposal data access interfaces that guarantee that the data access restrictions are always correctly followed. Moreover, the architecture is not dependant on one single data store, reducing vendor lock-in.

The potential benefits of using an architecture such as the S-DiSACA out-weight the earlier time investment into creating a simple API access interface and the flowcharts that use its operations. This is not only true during development, but also during the application maintenance and management as data access bugs are reduced and constrained to a specific set of classes.

TABLE 6.3: Test machine specifications.

OS	GNU/Linux
Kernel	5.3.0-40-generic #32~18.04.1-Ubuntu
CPU	Intel i5-6200 @ 2.30GHz
RAM	8GB
Motherboard	PS463E-07704KEP / TECRA A40-C (TOSHIBA)
Storage	ATA SAMSUNG MZNTY256
Other	Java OpenJDK 11.0.6 2020-01-14

In terms of the quantifiable performance key indicators, the time required by the architecture to generate and compile the Business Schema interfaces, connect to the Policy Server and the querying overhead are the measured, shown and analysed.

6.4.1 Testing Environment

To ensure that the results shown are reproducible, the specifications of the machine used to run the tests are shown in Table 6.3.

This machine hosted the entire architecture to remove network round trips from the results, as any network-related delays would be constant and simply added as many times as a message is sent through the network. As such, the assessment focuses on the performance of the architecture itself.

6.4.2 Connection and Interface Generation

The time required by the S-DiSACA to initialize before it can be used by the client application to issue data access requests was measured using the *nanoTime()* Java function. While this function does not allow to determine the current time, it allows to measure time intervals with nanosecond precision.

The architecture goes through five major steps during initialization:

1. Authenticate, where the client application establishes a secure communication channel to the Policy Server with mutual authentication;
2. GetMetadata, where the client application requests the flowcharts and operations it is allowed to execute;
3. Controller, where the sequence controller object is initialized with the metadata retrieved in the previous step;
4. Generate, where the ProxyAPI and Business Schema interfaces are generated;
5. Compile, where the ProxyAPI and Business Schema interfaces are implemented.

During the client application compilation phase, only steps 1, 2 and 4 are executed, while all five steps are carried out during runtime. To measure the amount of time each step takes, 10000 separate initializations were carried out. The results are shown in Table 6.4 and illustrated in Fig. 6.26.

TABLE 6.4: S-DiSACA initialization performance results.

	Authenticate	GetMetadata	Controller	Generate	Compile
Average	32,57 ms	4,71 ms	0,08 ms	1,12 ms	56,09 ms
Std. Dev.	20,41 ms	4,61 ms	0,05 ms	1,18 ms	41,96 ms
P_{95}	69,35 ms	15,50 ms	0,18 ms	3,05 ms	143,31 ms

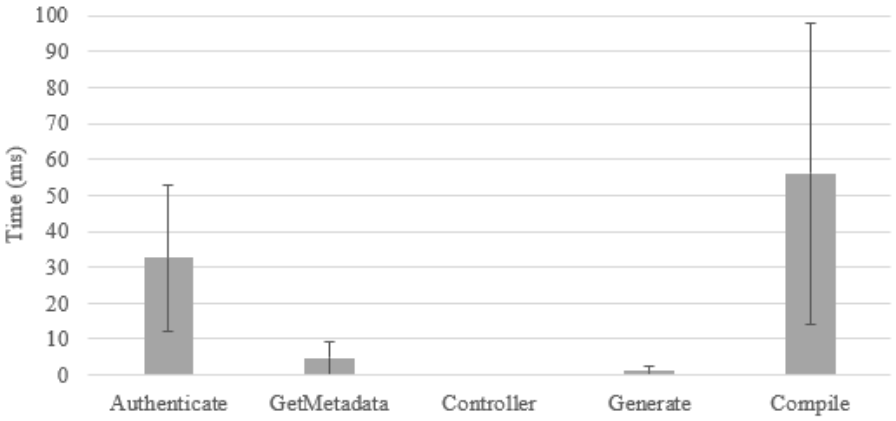


FIGURE 6.26: S-DiSACA initialization performance results.

Table 6.4 shows the average, the standard deviation and the 95^{th} percentile results for all five steps in milliseconds. It is clear from these results that the most expensive steps are the authenticate step, which uses asymmetric encryption, and the compile step that needs to use the Java Compiler. Regardless, the entire initialization takes $94,57$ ms during runtime (all steps) and just $48,4$ ms during compilation (steps 1, 2 and 4), on average. Even considering the 95^{th} percentile, the total time for the initialization remains below 250 ms. These results are acceptable since 1 second is considered to be the limit for no special feedback needed for the user [117].

6.4.3 Database Querying

So far the performance of the architecture during initialization was measured and analysed, which leaves the querying performance that the client application gets through it. Since it was noted that the communications between the client application and the policy server have several security features implemented, the overhead of these features was also measured. Table 6.5 and Fig. 6.27 show the results obtained.

It is clear from these results that the security features overhead adds approximately 9 ms to the processing time of each request, with the secure requests taking about $16,31$ ms and the unsecure requests $7,71$ ms on average. The overhead is actually less than 9 ms since the query execution time is factored into these values and adds to the query verification time. This is why the queries executed for this test were kept as simple as possible, as the verification time does not depend on the query complexity. Another conclusion that can be made from these results is that the 95^{th} percentile for both

TABLE 6.5: S-DiSACA querying performance results.

	Secure	Unsecure
Average	16,31 ms	7,71 ms
Std. Dev.	4,01 ms	2,79 ms
P_{95}	24,36 ms	12,73 ms
Throughput	61 queries/s	130 queries/s

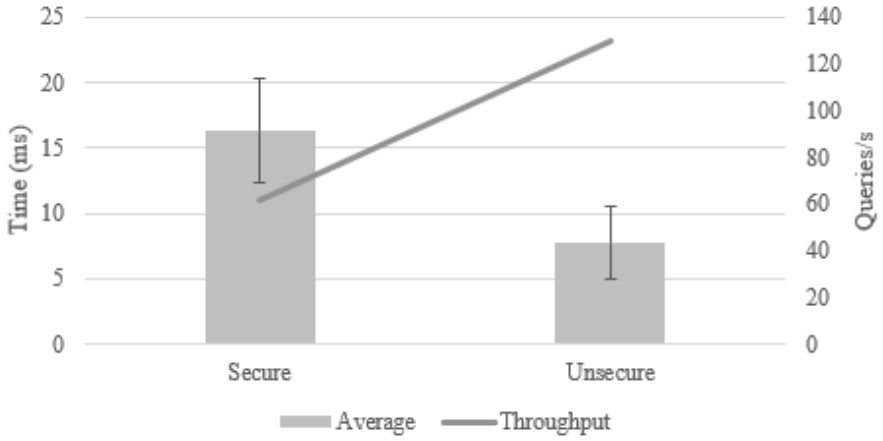


FIGURE 6.27: S-DiSACA querying performance results.

measurements is well below 0,1 seconds, the threshold after which users lose the feeling of operating directly on the data [117].

However, none of these results accounts for network round trip delays, which can increase these values above the 0,1 second limit (and lowers the impact of security features overhead). Fortunately, special feedback is only necessary if a request ever takes more than 1 second to complete. Therefore, client applications that use the S-DiSACA to access the data are still able to be used as interactive applications without frustrating the users with overly large delays.

6.4.4 Auditing

In this section, the results obtained by applying the policy correctness auditing algorithm from section 5.2 are shown.

To do so, a brute-force algorithm was first used that called the evaluation engine for every single possible combination of input values, and the decision to each combination was then compared to the decision outputted by the optimized algorithm, which only used the evaluation engine in the cases where the decision could not be predicted as detailed in chapter 5.2.

Furthermore, a proof of concept tool that implements the optimization algorithm for type-1 BDFIS was used and is available[2] along with the policies used here. The type-1 was chosen to demonstrate

[2]https://github.com/Regateiro/FuzzyAC/tree/master/java/BDFISAuditor

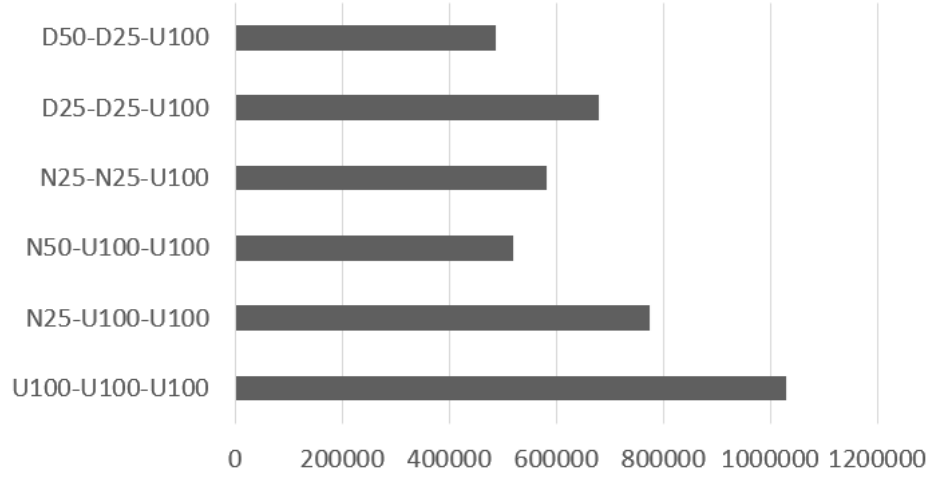


FIGURE 6.28: Number of calls to the evaluation engine given different policies.

empirically the correctness of the results shown in 5.1.6. Once again, this tool leverages the jFuzzyLogic Java library to implement fuzzy inference systems from FCL policy files.

Fig. 6.28 shows the number of calls done to the evaluation engine given distinct sets of three inputs variables (A, B, and C) with different linguistic terms that interact with one another in a carefully defined manner. Each variable is identified with a letter, which indicates a type of contribution, and a number that indicates the percentage of the domain that is classified by that contribution. The letter D stands for the *deny* contribution, the letter U for *unknown* contribution, and the letter N for contribution to *none*. Thus, the label "D25-D25-U100" would mean that the first and second variables contribute to the *deny* FDC in 25% of their domain (the rest is by default *unknown*) and the third variable always has an unknown contribution.

All three variables have a domain range of 101 values ($[0, 100]$), for a total of $101^3 = 1030301$ possible input value combinations. By applying Eq. 5.12 and given that the *unknown* factor (N_2 and R_2) cannot be optimized, if these three variables have a single range of *unknown* contribution spanning their entire domain, the number of calls is always $(101 + 0 + 0) * (101 + 0 + 0) * (101 + 0 + 0) = 101^3 = 1030301$. Thus, the "U100-U100-U100" case can be used to determine the degree of optimization obtained, since it always results in the worse-case scenario.

Another aspect that can be extracted from these results is that a variable with no contribution for a specific percentage of its domain translates directly to a reduction in the number of calls, since "N25-U100-U100" saw a 25% reduction in the number of calls to the evaluation engine, and "N50-U100-U100" saw a 50% reduction. The scenario labelled "N25-N25-U100", saw a reduction of 57% in the number of calls. It makes sense to be 57% since the algorithm only checks 75% of the domain of the first variable and then 75% of the second ($0.75 * 0.75 = 0.5625$). The scenarios using the *deny* contribution show that it is possible to optimize the auditing algorithm in these conditions. However, the efficiency of the optimization depends on the output of the system, as combinations can only be skipped once the output matches the contribution of the range.

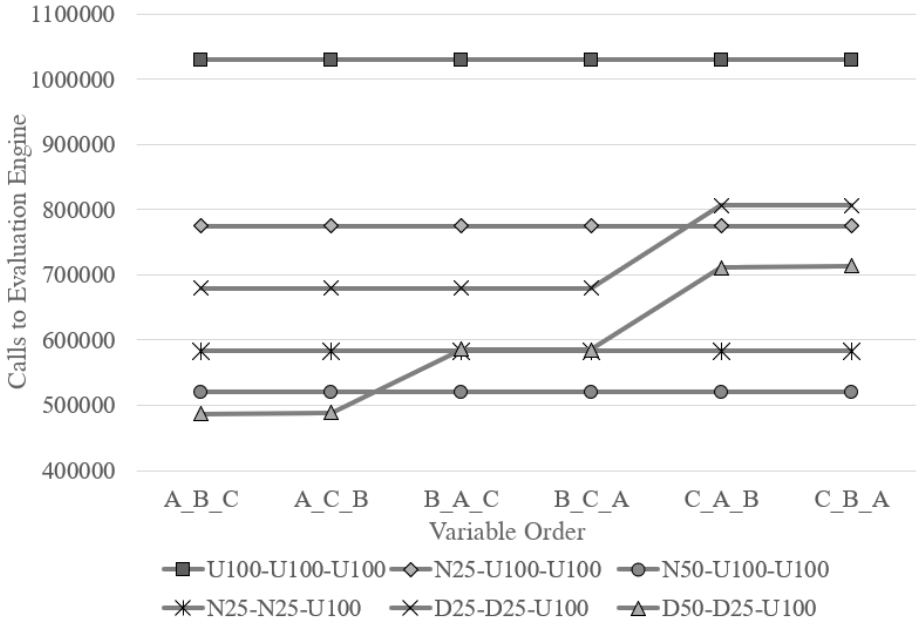


FIGURE 6.29: Number of calls to the evaluation engine given variable order permutations.

Fig. 6.29 shows how the order of the variables influenced the number of calls to the evaluation engine. Something that is immediately clear is that the ranges of no contribution do not impact the efficacy of the optimization algorithm by themselves. Only the scenarios with *deny* contribution ranges saw any change to the number of calls with different variable orders.

Looking at the scenarios with *deny* contribution ranges, it is clear that having the variables with these ranges at the start of the order produces the best results. It was also observed that the scenario "D50-D25-U100" always produced an equal or slightly better result after the last two variables where switched and the variable with the largest range of unknown contribution came last. For example, the order (A, B, C) and (A, C, B) had 487101 and 488351 calls to the evaluation engine, respectively. Furthermore, in the "D50-D25-U100" scenario the number of calls to the evaluation engine increased considerably in the (B, A, C) and (B, C, A) cases. This is because the variable with the greatest *deny* range size (A) was not the first in the order, causing the β_x factors to be smaller. In the (C, A, B) and (C, B, A) cases where the variable with the entire *unknown* contribution is the first in the order, the number of calls increased considerably for the same reason. This matches the expected behaviour described in section 5.2.

Finally, a previously created policy that was not constructed specifically to test the algorithm was used to verify the effectiveness of the algorithm in more realistic conditions. The policy in question encoded a single vague concept, the *Expertise* of a subject, from the number of publications (NoP) and the number of citations (NoC) related to said subject in a specific area. Depending on the level of expertise if the subject it could be granted or denied access to a resource, such as being able to edit a Wikipedia article.

TABLE 6.6: Input variables range partitioning.

Variable	Domain Range	Read	Write
NoP	$[3, 5]$	Unknown	Unknown
NoP	$[5, 10]$	No	No
NoP	$[10, 15]$	Unknown	Unknown
NoP	$[15, 18]$	Unknown	Grant
NoC	$[0, 4]$	No	No
NoC	$[4, 10]$	Grant	Unknown
NoC	$[10, 30]$	No	No
NoC	$[30, 40]$	Grant	Unknown
NoC	$[40, 80]$	Unknown	Unknown
NoC	$[80, 100]$	Grant	Unknown

Table 6.6 shows each input variable already partitioned into classified domain ranges. From it, it can be seen that the domain range of the NoP variable is $[3, 18]$ and the domain range of the NoP variable is $[0, 100]$.

The table shows how common ranges of no contribution can be. About 38% of the NoP variable domain and 26% of the NoC variable domain have no associated contribution to the output. This can be explained by the fact that different membership functions usually do not change their output when one plateaus, as shown in Fig. 5.4.

This relies on the assumption that the membership functions plateau at some point. Functions that do not plateau (and are triangular in shape, instead of trapezoidal) have fewer ranges of no contribution as a result. However, not every membership function changes its output in the same domain ranges. Thus, it can be expected that triangular-shaped functions result in more *deny* or *grant* contribution ranges instead.

Fig. 6.30 shows the number of calls made to the evaluation engine for both the *Read* and *Write* permissions that were defined in the FCL. The *Max* line indicates the number of calls that a brute-force algorithm would perform.

The best result obtained was for the *Read* permission, where only 499 calls out of 1616 were made to the evaluation engine (31%). Similarly, the *Write* permission required 736 calls out of 1616 (46%).

Finally, to further show the applicability and efficacy of this algorithm on different scenarios, the risk-based policy described in [43] (which was discussed in section 3.1.2) was written and passed through the algorithm. The three input variables all possess the same domain $[0, 1]$, which means that with an update step of 0.01 a brute force algorithm would require $(101 + 0 + 0) * (101 + 0 + 0) * (101 + 0 + 0) = 101^3 = 1030301$ evaluations to complete the analysis by applying Eq. 5.12. However, while the membership functions and rules that apply to each of the three input variables are defined by the authors, no output permissions were specified nor how to translate the output level of risk into which permissions are granted or denied. This was defined for the algorithm as shown in Tab. 6.7.

Thus, permissions P1 to P4 were defined in order of importance for the scenario, meaning that as risk increases, the first permission to be revoked is P4, then P3, etc. Fig. 6.31 shows the assessment of the algorithm in this scenario. The algorithm was able to reduce the number of calls to the evaluation

TABLE 6.7: Output rules in the risk-based policy.

Risk\Perm.	P1	P2	P3	P4
Negligible	Grant	Grant	Grant	Grant
Low	Grant	Grant	Grant	Deny
Moderate	Grant	Grant	Deny	Deny
High	Grant	Deny	Deny	Deny
Unacceptable High	Deny	Deny	Deny	Deny

engine considerably, using between 6.59% and 8.42% of the number of evaluation calls that the brute force algorithm would require for all permissions.

While these results are promising, the efficacy of the algorithm still depends greatly on the policy itself. The unknown contribution ranges depend on how much the membership function ranges that contribute to a single Fuzzy Decision Component (FDC) overlap and where they overlap. Additionally, fuzzy rules that take the same linguistic terms and apply them to different output outcomes can also contribute to the increase of unknown contribution ranges. Therefore, additional work into further optimizing this algorithm is necessary to combat scenarios that exhibit these characteristics.

6.5 Architecture Literature Positioning

With the S-DiSACA implementation, configuration, usability, evaluation and performance assessment detailed, the architecture is now in a position where it can be properly compared with the related works and technologies found in the state of the art.

In terms of the remote access to databases, the architecture can be compared to solutions such as database drivers like Java Database Connectivity (JDBC) and Object-Relational Mapping (ORM) tools

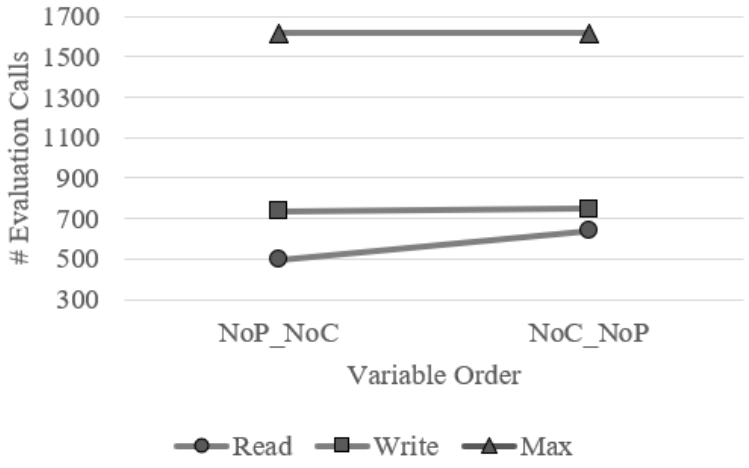


FIGURE 6.30: Number of calls to the evaluation engine.

like Hibernate. While JDBC provides developers with the highest levels of flexibility by allowing them to execute any query on the database, Hibernate maps the tables to classes and each entry on the table becomes instances of those classes. This allows developers to let Hibernate deal with the database queries while they work on the objects.

However, neither of these solutions have out-of-the-box communication security, protect the database credentials or are able to hide the database schema from being accessible from the client application. S-DiSACA supports all this out-of-the-box, but it does restrict the developers flexibility to execute what they want on the database. They are restricted not only to the operations that are defined in the data access APIs, but they must also follow the operation sequencing to ensure that the use cases are properly implemented. The architecture also generates data access interfaces on the client application for the developers to use. These interfaces prevent costly development and quality assurance time being spent finding and correcting errors, but they require some effort to be spent configuring the architecture properly for which some configuration tools already exist.

Regarding the soft access control model, the BDFIS is a modified Mamdani-type Fuzzy Inference System (FIS) where the output layer is configured with permission variables, each with a deny and a grant fuzzy set that allows to output an access control decision once defuzzified. Moreover, its policies can be audited by a security expert using the optimized auditing algorithm researched for FIS with binary outputs.

Most of the soft access control models found in the literature focus on specific application scenarios while the BDFIS is a general solution. This means that while the literature solutions are easier to fine-tune to their respective application scenarios by already having the necessary input/output variables, fuzzy sets, membership functions and fuzzy rules defined, the BDFIS requires some more effort to achieve the same level of fine-tuning. In contrast, the BDFIS can be used in any access control scenario as it is not designed with any specific set of variables, rules or vague concepts in mind. Table 6.8 shows the comparison between BDFIS and the related works in the literature.

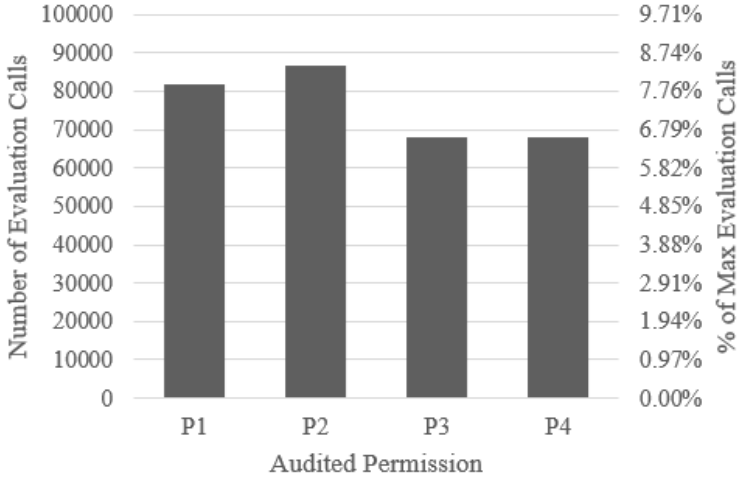


FIGURE 6.31: Evaluation calls made by the optimized algorithm in the risk scenario.

TABLE 6.8: Soft access control models comparison.

	BDFIS	[44]	[41]	[23]	[43]	[77]
Concept	Any	Risk	Roles	Trust	Risk	Trust
Inputs	Any	3	1	3	3	3
Outputs	Any	1	Any	1	1	1

Most of the works in the literature seem to take the approach of defining three input variables that are used to calculate the level of either risk or trust, measures that are used to determine which permissions and levels of access to give to the users and other devices. The only work that stands out is [41] which adapts the Role-Based Access Control (RBAC) to use fuzzy logic by fuzzifying the subject-role and role-permission relations. This means that the model makes its access control decisions solely based on the roles of the subjects, but on the other hand it allows to have any number of permissions granted or denied as an output.

The architecture as a whole trades some of the flexibility that other state of the art solutions provide for ease of development in the case of the remote database access, and for wider applicability in the case of the soft access control model. Both of these trade-offs were made to broaden the application scenarios of the S-DiSACA and to bring an access control solution to scenarios where none existed before, such as vandalism prevention for the Wikipedia.

6.6 Summary

In this chapter, the S-DiSACA and the previously presented auditing algorithm were evaluated in terms of their correctness and performance. A proof of concept of the architecture and a standalone policy auditing tool was built to obtain the results shown and made available in a code repository. Moreover, the tools and database schemas developed to configure and support the architecture were also shown. This led to comparing the initial configuration cost of the architecture versus the development time gained from the automation and lower quality assurance needs. Potential new venues for configuration tools were also discussed. A comparison of the S-DiSACA against the related works in the literature was also carried out.

During the correctness evaluation, the S-DiSACA proof of concept was used in a variety of scenarios that attempted to bypass several of the security requirements brought forth in section 1.1. The performance evaluation consisted of measuring and analysing the time required to generate and compile the Business Schema interfaces, connect to the Policy Server and query the data. The results showed that the enhanced security features did not impact the performance in a way that made the architecture unusable in interactive applications.

Regarding the policy correctness auditing algorithm, several test policies were used to showcase the expected behaviour in terms of performance with different ratios of no, single and unknown contribution ranges. The test scenario with the subject *Expertise* was also tested, along with a policy adapted from a scenario found in the literature.

Chapter 7

Discussion and Conclusions

This thesis focused on two fronts to broaden the application scenarios of access control: the development of secure and correct database applications; and the applicability of access control systems in scenarios where traditional solutions are not a good fit.

In section 1.1, several issues related to these contexts were identified and a list of research goals was introduced to solve them. These goals are discussed to determine to which extent they were achieved if any new issues were discovered, and what possible lines of research can follow.

As previously mentioned, goals 1 to 6 had already been achieved in the previous iteration of this work, the Secure, Dynamic and Distributed Role-based Access Control Architecture (S-DRACA). However, several issues remained that were addressed in this research work. These issues were transposed into research questions in section 1.3, which were answered over the course of the dissertation.

7.1 Interface Generation and Implementation

The first goal was intended to ease the application development effort by having a tool automatically request the authorized operations and generate data access interfaces tailored to each application, which is closely tied to RQ3. This way, developers are not required to master which operations they are allowed to execute on the data and do not have to implement the necessary code to execute each operation either. This reduces the time spent developing and testing the correctness of the application. Not only that, changes to the policies that define the authorized operations for an application should also be reflected automatically to let developers know what must be changed in the application code and where.

This is the initial goal that led to the research and development of the first architecture and has evolved ever since up to the previous iteration. Since it is a very mature concept, the core idea in the Secure, Dynamic and Distributed Soft Access Control Architecture (S-DiSACA) remained the same. However, the process had to be adjusted due to the interface changes incurred from the need to support non-relational databases. Further changes were also necessary to support the sixth goal to ensure that the generated interfaces allowed developers to follow the defined flowcharts with minimal hassle.

Nonetheless, this concept can still be improved in the future. Developers still need to code the execution of each operation within a flowchart so that the parameters can be set. An evolution of this approach could have a single Business Schema with the various flowcharts defined within, allowing the developers to request the execution of a flowchart instead of each operation. To handle the parameters, the observer pattern could be used to have the generated Business Schema request the necessary parameters and pass the intermediate results to a listener.

7.2 Operation and Parameter Protection

The second and third goals are also tied to RQ3 and focus on ensuring that the operations are always executed as intended and with valid parameters. Standard solutions such as Java Database Connectivity (JDBC) do not support this level of security, allowing users to potentially execute arbitrary operations or to modify the parameters used via reflection.

The previous iteration took an initial step to address these issues. Instead of sending the Structured Query Language (SQL) query associated with an operation, a token that identifies the operation was sent instead. Then, the Business Schemas on the client application would call a stored procedure, passing the token and the parameters. This stored procedure would then load the correct SQL query and execute it. While this approach prevented users from executing arbitrary operations, it was dependent on relational database features such as stored procedures. Furthermore, the stored procedure was implemented for a specific relational database and it needed to be adapted to work in others.

The approach presented in this dissertation turns the idea of sending a token to the server into a more traditional web server architecture. The client application sends a request for a specific operation to be executed on the data, and the Policy Server executes it using the custom application programming interface (API) developed for the application scenario. Since the operation protection is based solely on having the operations defined on the server so that the client cannot modify them, the S-DiSACA does not allow the custom API to be modified at runtime. However, such a feature might be interesting as a future goal to allow new operations to be added or to quickly fix a bug in the implementation of the API without stopping the Policy Server and affecting the existing client applications.

Finally, protecting the parameters used in the operations is a much harder goal to achieve. The previous iteration made it so protected parameters needed to be passed using the Business Schemas. The client application would read the data in a Business Schema until the row with the desired data was selected and then it would be passed to the next Business Schema for execution. While this approach would prevent parameters from being passed directly to the Business Schemas, it did not validate the parameters on the server-side, allowing reflection mechanisms to be used. However, this approach provided the framework to the solution presented in this dissertation where a signed object from the Policy Server would be used to select the desired parameter, which can be validated by the server.

Unfortunately, the layers of security added to the replies that the client application receives from the Policy Server does impact the performance. Optimizing the security material and the validation checks used on every request is a possible step moving forward to target application scenarios with tighter performance requirements.

7.3 Database Credentials and Secure Communication

The fourth goal focused on solving a particular issue with current client database applications used within businesses, which is that the applications tend to have the database credentials included in them. Although these applications are not meant to be used by the public, they may still be found in semi-public contexts such as a reception desk. If the employee responsible for that area needs to leave for some reason or is coerced to insert a device in the host machine such as a USB drive, the client application can be targeted and the database credentials extracted. This goal is therefore connected with RQ2.

To prevent this from ever being possible, the previous iteration focused on pushing the database credentials to a proxy server on the server-side. The client application would connect and authenticate with this proxy server, which would then connect to the database and relay the communication. However, this approach suffered from an issue where if the proxy server had a vulnerability and was exploited, the database credentials would still be obtained. This idea was built upon so that two servers are now involved, and to obtain the database credentials both servers would have to be exploited. The search for further vulnerabilities should be carried out in the future to mitigate potential exploits.

The fifth goal is intended for specialized scenarios where placing trust in certificate authorities or paying for a digital certificate is not acceptable. Since communication encryption in languages such as Java is implemented requiring such certificates, having a secure communication channel in these conditions is made difficult. The previous iteration implemented secure communication channels using unauthenticated Transport Layer Security (TLS) channels and then modifying the agreed key with a pre-shared key, which would indicate that both endpoints know the shared secret if they could communicate after. This process would also serve to authenticate the client and the server. However, if the shared secret was ever guessed, man-in-the-middle attacks could be carried out. This issue was addressed by using asymmetric keys, allowing the client application to send an ephemeral random key to the server to function as a shared secret. The server would be authenticated via the public key used by the client to send the ephemeral key, and the client could then authenticate itself by sending its credentials.

Unfortunately, both approaches rely on reflection mechanisms to modify the initial TLS key to work, meaning that the current implementation relies on the vendor-specific implementation of the TLS objects made by Oracle for Java 8. As such, proper implementation of the TLS protocol that allows using pre-shared keys should be adopted or created in the future.

7.4 Operation Sequencing

The sixth goal is one of the most important goals in the architecture, as it aims to allow security experts to define use cases through a sequence of operations that can be executed by the client application. This goal is therefore linked with the second half of RQ3.

The previous iteration allowed the definition of simple sequences of operations. These were quite restrictive because from one operation only one other operation could follow it. There was no ability to add branches and choices to the execution. Furthermore, the sequence controller that checked if the sequences were being followed during the execution of operations could be bypassed using reflection mechanisms. The current iteration, on the other hand, supports the definition of these sequences using graphs (known as flowcharts) that are now also validated on the server-side. Furthermore, the Business Schemas implemented on the client-side reflect these flowcharts and ensure that developers cannot unwillingly execute operations out of order. This effectively allows the use case logic to be implemented automatically into the application code and promotes the importance of proper use case definition early on in the application development life-cycle.

While a command-line tool to generate these flowcharts exists, a more intuitive tool that not only allows to generate these sequences but also shows them as they are being manipulated is essential to be pursued in the future. Furthermore, when a flowchart is modified, a transition period where the old flowcharts would still be accepted to allow developers to provide an update to the client application could be beneficial.

7.5 Interface Abstraction for Generic APIs

The seventh goal aims to resolve an applicational gap that leans heavily on RQ1. This gap formed due to the previous iterations being focused solely on relational database applications while non-relational databases became more ubiquitous. The interface generation model used was based on SQL queries (through JDBC) which is a common language between different relational databases. Non-relational databases, however, have varying interfaces and specificities that make it difficult to have a single common API. Thus, being able to define and register with the Policy Server a custom data access API that is built to make the most out of these non-relational databases.

The previous iteration supported this goal by generating and implementing interfaces that wrapped a JDBC connection object. Thus, it allowed executing create, read, update, and delete (CRUD) operations over this object while providing the application with a subset of the standard JDBC API functions. However, this came with a cost: the client applications were required to have a direct connection to the database. A solution using a proxy server was developed that partially solved this issue, but the client application still had to connect to the database to instantiate the JDBC object. Furthermore, the proxy server relied on reflection mechanisms to relay the communication, so distinct versions of Java could break the solution.

To gracefully address these issues the interfaces being generated cannot rely on JDBC due to its limitations. In the S-DiSACA, the generated interfaces request the execution of the operations to the Policy Server using messages instead of an underlying JDBC object, granting more freedom regarding

how the data access API is defined and implemented. However, this approach also comes with a cost. The previous approach used a JDBC object because it is a common method to connect and manipulate data in relational databases in Java, and as such it provided a consistent API that the developers did not have to relearn. While the new approach can be used to generate and implement a JDBC compatible API if desired even if the data is not stored in a relational database, it now needs to be manually defined.

Future work focused on harmonizing both approaches where a tool can generate and implement the Policy Server API based on well-known data access mechanisms, for both relational and non-relational data stores, could close this gap and help security experts to spend less time on the definition of the data access API to be used by the client applications. Another aspect worth considering for a possible future work is related to synchronous operation executions. Some operations may update database records and if the database does not support transactions the functionality should be incorporated into the S-DiSACA. Furthermore, if the architecture is scaled horizontally and multiple instances are running at the same time, they will need to coordinate the execution of these operations as well. Then, a study of the cost of the application of this synchronization across instances of the architecture should be carried out.

7.6 Dynamic Permissions and Soft Requirements

The eighth goal sets itself apart from the previous goals as it involves a new line of research that had not been followed before within this work, which is why RQ4, RQ5 and RQ6 connect to it. The idea of allowing permissions to be dynamically assigned to users as they issue data access requests expand the applicability of the architecture to scenarios where there is no fixed set of users. Furthermore, the approached taken to support this goal also included adding support for soft access control requirements to scenarios where it is difficult to crisply define who has access to which data given some information about the user.

This approach has shown to have many benefits. It has allowed policies to be easily modified at runtime using an existing standard language (RQ4), something that the previous iteration of the architecture lacked; a new user can now make a data access request as long as the necessary information about him is available from a trusted source; and the access control rules can the written using fuzzy logic, which makes it easier for new security experts to understand its purpose.

Unfortunately, this dynamism and abstract access control rules came with some great caveats: first, the cost of configuring this must not outweigh the benefits obtained during development; and second, auditing the correctness of the access control policies is extremely difficult. The first caveat is addressed by presenting the tools used to ease configuration and by providing future venues for new and better tools. Regarding the second caveat, since there is no set of users allowed to access the data, the full domain of the parameters used to grant permissions to the users can be combined at each request. Ensuring that the access control policy allows only expected combinations of input values to have certain permissions is impossible to do without a specialized tool to analyse the policies.

The first step towards creating this auditing tool was researched and presented here. With it, security experts can now analyse the clusters of input parameter values that are granted permissions and determine if they are no strange outliers (RQ5). While the performance results for this tool and its associated search algorithm are promising, there are several avenues for further optimization in the future that also shine a light into how variables interact with each other within these systems, including using formal verification of the inference chains in the policies to address any conflicting rules that may cause ranges with unknown decision support. Furthermore, the fuzzy inference systems used to incorporate the soft access control requirements can also be used to integrate artificial intelligence systems. Instead of requiring a security expert to define the membership functions, an artificial intelligence system can provide the membership degrees to each linguistic term instead. This approach has many nuances and is also a viable line of research.

Finally, several questions about the applicability of fuzzy logic in access control have also been answered. Fuzzy logic is by nature vague, leading to the auditing issue discussed above, and thus using it to protect access to sensitive information is a questionable approach (RQ6). The point is that fuzzy logic should be used in scenarios where traditional access control models are not a good fit, such as community-managed information (e.g. Wikipedia) and systems with no set list of users (e.g. communication between Internet of Things (IoT) sensors).

Appendix A

Example Policy File Using FCL

In this appendix, an example policy definition for the BDFIS using FCL is shown and discussed. To demonstrate how the abstract layers can be handled, this example focuses on a type-1 BDFIS.

LISTING A.1: First function block in example FCL policy.

```
1  FUNCTION_BLOCK VariableInference
2
3      VAR_INPUT
4          NoP             :  REAL; (* RANGE(0 ..) *)
5          NoC             :  REAL; (* RANGE(0 ..) *)
6      END_VAR
7
8      VAR_OUTPUT
9          Expertise       :  REAL;
10     END_VAR
11
12     FUZZIFY NoP
13         TERM Low        := (3, 1) (5, 0) ;
14         TERM Medium     := (3, 0) (5, 1) (13, 1) (18, 0) ;
15         TERM High       := (10, 0) (15, 1) ;
16     END_FUZZIFY
17
18     FUZZIFY NoC
19         TERM Low        := (0, 1) (4, 1) (10, 0) ;
20         TERM Medium     := (4, 0) (10, 1) (40, 1) (80, 0) ;
21         TERM High       := (30, 0) (100, 1) ;
22     END_FUZZIFY
23
24     DEFUZZIFY Expertise
25         TERM Low        := 1 ;
26         TERM Medium     := 2 ;
27         TERM High       := 3 ;
28         TERM Very_High  := 4 ;
29         METHOD          : COGS ;
30     END_DEFUZZIFY
31
32     RULEBLOCK Expertise
33         RULE 0: IF (NoP IS Low)    AND (NoC IS Low)    THEN Expertise IS Low;
34         RULE 1: IF (NoP IS Low)    AND (NoC IS Medium) THEN Expertise IS High;
35         RULE 2: IF (NoP IS Low)    AND (NoC IS High)   THEN Expertise IS Very_High;
36         RULE 3: IF (NoP IS Medium) AND (NoC IS Low)    THEN Expertise IS Low;
37         RULE 4: IF (NoP IS Medium) AND (NoC IS Medium) THEN Expertise IS Medium;
38         RULE 5: IF (NoP IS Medium) AND (NoC IS High)   THEN Expertise IS High;
39         RULE 6: IF (NoP IS High)   AND (NoC IS Low)    THEN Expertise IS Low;
40         RULE 7: IF (NoP IS High)   AND (NoC IS Medium) THEN Expertise IS Medium;
41         RULE 8: IF (NoP IS High)   AND (NoC IS High)   THEN Expertise IS Very_High;
42     END_RULEBLOCK
43
44 END_FUNCTION_BLOCK
```

Listing A.1 shows the first half of the FCL definition, where the first set of rules that maps the input layer to the abstract layer variables and linguistic terms. It begins by defining a FUNCTION_BLOCK called *VariableInference* in line 1, which is closed in line 44. A function block is an FCL feature that

contains the variable and rules definitions. The input variables *NoP* and *NoC* are defined in the VAR_INPUT block in lines 3 to 6. Likewise, the output variable *Expertise* (for this function block) is defined in the VAR_OUTPUT block in lines 8 to 10. This output variable is an abstract variable that is used as input for the next function block.

The step that follows is the input variable fuzzification process. This process for each of these variables is defined in the FUZZIFY blocks in lines 12 to 16 and 18 to 22. These FUZZIFY blocks have TERM features that define the linguistic terms for each variable, including their names and a function definition. These functions can be defined using a set of (x, y) points for piecewise linear functions or a single x value for a singleton function.

The DEFUZZIFY block in lines 24 to 30 defines the output variable linguistic terms and the defuzzification method to use. Since this is actually the definition of the abstract variable that is used in the next function block, singleton functions are used to simplify the process. The RULEBLOCK in lines 32 to 42 defines the set of rules to apply between the input and output linguistic terms, one per RULE entry.

LISTING A.2: Second function block in example FCL policy.

```
1  FUNCTION_BLOCK AccessControl
2
3      VAR_INPUT
4          Expertise   :   REAL; (* RANGE(0 ..) *)
5      END_VAR
6
7      VAR_OUTPUT
8          Read        :   REAL;
9          Write       :   REAL;
10     END_VAR
11
12     FUZZIFY Expertise
13     END_FUZZIFY
14
15     DEFUZZIFY Read
16         TERM Deny   :=  0 ;
17         TERM Grant  :=  1 ;
18         METHOD      : COGS ;
19     END_DEFUZZIFY
20
21     DEFUZZIFY Write
22         TERM Deny   :=  0 ;
23         TERM Grant  :=  1 ;
24         METHOD      : COGS ;
25     END_DEFUZZIFY
26
27     RULEBLOCK Read
28         RULE 0: IF (Expertise IS Low)       THEN Read IS Deny;
29         RULE 1: IF (Expertise IS Medium)    THEN Read IS Grant;
30         RULE 2: IF (Expertise IS High)      THEN Read IS Grant;
31         RULE 3: IF (Expertise IS Very_High) THEN Read IS Grant;
32     END_RULEBLOCK
33
34     RULEBLOCK Write
35         RULE 0: IF (Expertise IS Low)       THEN Write IS Deny;
36         RULE 1: IF (Expertise IS Medium)    THEN Write IS Deny;
37         RULE 2: IF (Expertise IS High)      THEN Write IS Deny;
38         RULE 3: IF (Expertise IS Very_High) THEN Write IS Grant;
39     END_RULEBLOCK
40
41 END_FUNCTION_BLOCK
```

The second half of the policy definition is shown in Listing A.2 and the function block defined within follows the same structure as the first. However, the VAR_INPUT block declares the *Expertise* abstract variable as input instead. The associated FUZZIFY block in lines 7 to 10 is left empty, as the

linguistic terms and their associated rule strengths are obtained from the first function block and set programmatically.

Since this is the last function block, it maps the abstract variables defined in the previous one to the output variables. These output variables are the permissions that must be either granted or denied, and the VAR_OUTPUT block defines the *Read* and *Write* example permissions. Moreover, these output variables use the predefined FDC linguistic terms, declared in the DEFUZZIFY blocks in lines 15 to 19 and 21 to 25. Note that the *Deny* term is set to the x value 0, and the *Grant* term to x value 1.

Finally, one RULEBLOCK per output variable is defined in lines 27 to 32 and 34 to 39, stating whether the *Read* and *Write* permissions are granted or denied depending on the *Expertise* level. This FCL definition can be loaded and used out of the box by tools such as jFuzzyLogic [118], [119] in Java to implement a BDFIS, requiring only that the program sets the abstract variable linguistic terms and their membership degrees in the FUZZIFY blocks as they are used in new function blocks.

Bibliographic References

[1] R. Shirey, "Internet Security Glossary, Version 2", Tech. Rep. 9, Aug. 2007, pp. 1278–1308. DOI: 10.17487/rfc4949. [Online]. Available: https://www.rfc-editor.org/info/rfc4949.

[2] Oracle, *JDBC Introduction*, 1997. [Online]. Available: http://docs.oracle.com/javase/tutorial/jdbc/overview/index.html (visited on 11/11/2019).

[3] Microsoft, *LINQ (Language-Integrated Query)*, 2007. [Online]. Available: http://msdn.microsoft.com/en-us/library/bb397926.aspx (visited on 11/11/2019).

[4] Microsoft, *ADO.Net*. [Online]. Available: http://msdn.microsoft.com/en-us/library/e80y5yhx(v=vs.110).aspx (visited on 04/28/2014).

[5] C. Bauer and G. King, *Hibernate in Action*. 2005, p. 408, ISBN: 193239415X. [Online]. Available: https://profs.info.uaic.ro/~ogh/files/doc/Manning%20-%20Hibernate%20In%20Action.pdf.

[6] A. B. M. Moniruzzaman and S. A. Hossain, "NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison", *International Journal of Database Theory and Application*, vol. 6, no. 4, pp. 1–14, Jun. 2013. arXiv: 1307.0191.

[7] R. P. Padhy, M. R. Patra, and S. C. Satapathy, "RDBMS to NoSQL: Reviewing Some Next-Generation Non-Relational Database's", *International Journal of Advanced Engineering Sciences and Technologies*, vol. 11, no. 11, pp. 15–30, 2011, ISSN: 2230-7818.

[8] N. Leavitt, "Will NoSQL Databases Live Up to Their Promise?", *Computer*, vol. 43, no. 2, pp. 12–14, 2010. DOI: 10.1109/MC.2010.58.

[9] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters", *Communications of the ACM*, vol. 51, no. 1, p. 107, 2008, ISSN: 10782. DOI: 10.1145/1327452.1327492.

[10] T. White, *Hadoop: The Definitive Guide*, 4th ed. O'Reilly Media, 2015, vol. 54, p. 756, ISBN: 978-1491901632.

[11] M. Dayalan, "MapReduce: Simplified Data Processing on Large Cluster", *International Journal of Research and Engineering*, 2018, ISSN: 23487852. DOI: 10.21276/ijre.2018.5.5.4.

[12] A. Halfaker, Y. Panda, A. Sarabadani, J. Du, and A. Wight, *Objective Revision Evaluation Service*. [Online]. Available: https://ores.wikimedia.org/ (visited on 05/16/2019).

[13] Ó. M. Pereira, R. L. Aguiar, and M. Y. Santos, "CRUD-DOM: A Model for Bridging the Gap between the Object-Oriented and the Relational Paradigms", in *Proceedings - 5th International Conference on Software Engineering Advances, ICSEA 2010*, Nice: IEEE, Aug. 2010, pp. 114–122, ISBN: 9780769541440. DOI: 10.1109/ICSEA.2010.25. [Online]. Available: http://ieeexplore.ieee.org/document/5615022/.

[14] Ó. M. Pereira, R. L. Aguiar, and M. Y. Santos, "CRUD-DOM: A Model for Bridging the Gap Between the Object-Oriented and the Relational Paradigms - an Enhanced Performance AssessmentBased on a Case Study", *International Journal On Advances in Software*, vol. 4, no. 1, pp. 158–180, 2011. [Online]. Available: https://hdl.handle.net/10773/7959.

[15] Ó. M. Pereira, R. L. Aguiar, and M. Y. Santos, "ACADA: Access Control-driven Architecture with Dynamic Adaptation", *SEKE'12 - 24th Intl. Conf. on Software Engineering and Knowledge Engineering*, pp. 387–393, 2012. [Online]. Available: http://repositorium.sdum.uminho.pt/handle/1822/19866.

[16] Ó. M. Pereira, "DACA: Architecture to Implement Dynamic Access Control Mechanisms on Business Tier Components", PhD thesis, University of Aveiro, 2013. [Online]. Available: https://hdl.handle.net/10773/11966.

[17] D. J. R. Figueiral, "Arquitetura Dinâmica de Controlo de Acesso", Master's thesis, 2012. [Online]. Available: https://hdl.handle.net/10773/10892.

[18] D. D. Regateiro, "A secure, distributed and dynamic RBAC for relational applications", Master's thesis, University of Aveiro, 2014, p. 144. [Online]. Available: http://hdl.handle.net/10773/14045.

[19] W. Zeng, Y. Yang, and B. Luo, "Content-Based Access Control: Use data content to assist access control for large-scale content-centric databases", *Proceedings - 2014 IEEE International Conference on Big Data, IEEE Big Data 2014*, pp. 701–710, 2015. DOI: 10.1109/BigData.2014.7004294.

[20] M. Pelc, "Context-aware Fuzzy Control Systems", *International Journal of Software Engineering and Knowledge Engineering*, vol. 24, no. 5, pp. 825–856, Jun. 2014, ISSN: 0218-1940. DOI: 10.1142/S0218194014500326.

[21] A. Chen *et al.*, "A dynamic risk-based access control model for cloud computing", in *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)*, IEEE, Oct. 2016, pp. 579–584, ISBN: 978-1-5090-3936-4. DOI: 10.1109/BDCloud-SocialCom-SustainCom.2016.90.

[22] R. McGraw, "Risk-Adaptable Access Control", *Privilege (Access) Management Workshop. NIST - National Institute of Standards and Technology - Information Technology Laboratory*, pp. 3-17–3-18, 2009. DOI: 10.6028/NIST.SP.800-95.

[23] P. N. Mahalle *et al.*, "A Fuzzy Approach to Trust Based Access Control in Internet of Things", in *Wireless VITAE 2013*, IEEE, Jun. 2013, pp. 1–5, ISBN: 978-1-4799-0239-2. DOI: 10.1109/VITAE.2013.6617083.

[24] Ó. M. Pereira, D. D. Regateiro, D. J. Simões, and R. L. Aguiar, "A Literature Review of Access Control Mechanisms for the SQL Standard", in *Handbook of Research on Innovations in Access Control and Management*, A. K. Malik, A. Anjum, and B. Raza, Eds., 2015.

[25] Ó. M. Pereira, V. Semenski, D. D. Regateiro, and R. L. Aguiar, "The XACML Standard - Addressing Architectural and Security Aspects", in *Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security*, Porto: SCITEPRESS - Science and Technology Publications, 2017, pp. 189–197, ISBN: 978-989-758-245-5. DOI: 10.5220/0006224901890197.

[26] Ó. M. Pereira, D. D. Regateiro, and R. L. Aguiar, "Extending RBAC Model to Control Sequences of CRUD Expressions", *SEKE'14 - Intl. Conf. on Software Engineering and Knowledge Engineering*, vol. 2014-Janua, no. January, pp. 463–469, 2014, ISSN: 2325-9000. [Online]. Available: https://hdl.handle.net/10773/12568.

[27] Ó. M. Pereira, D. D. Regateiro, and R. L. Aguiar, "Role-Based Access Control Mechanisms", in *2014 IEEE Symposium on Computers and Communications (ISCC)*, vol. 2, Vancouver: IEEE, Jun. 2014, pp. 1–7, ISBN: 978-1-4799-4277-0. DOI: 10.1109/ISCC.2014.6912546.

[28] Ó. M. Pereira, D. D. Regateiro, and R. L. Aguiar, "Secure, Dynamic and Distributed Access Control Stack for Database Applications", *International Journal of Software Engineering and Knowledge Engineering*, vol. 25, pp. 364–369, Nov. 2015, ISSN: 0218-1940. DOI: 10.18293/SEKE2015-049.

[29] F. Fradique Duarte, D. D. Regateiro, Ó. M. Pereira, and R. L. Aguiar, "On the Prospect of using Cognitive Systems to Enforce Data Access Control", in *Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security*, SCITEPRESS - Science and Technology Publications, 2017, pp. 412–418, ISBN: 978-989-758-245-5. DOI: 10.5220/0006370504120418.

[30] D. D. Regateiro, Ó. M. Pereira, and R. L. Aguiar, "On the Application of Fuzzy Set Theory for Access Control Enforcement", in *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications*, vol. 4, SCITEPRESS - Science and Technology Publications, 2017, pp. 540–547, ISBN: 978-989-758-259-2. DOI: 10.5220/0006469305400547.

[31] D. D. Regateiro, Ó. M. Pereira, and R. L. Aguiar, "SeqBAC: A Sequence-Based Access Control Model", in *Proceedings of the 30th International Conference on Software Engineering and Knowledge Engineering*, vol. 2018, Jul. 2018, pp. 276–319. DOI: 10.18293/SEKE2018-099.

[32] D. D. Regateiro, Ó. M. Pereira, and R. L. Aguiar, "BDFIS: Binary Decision Access Control Model Based On Fuzzy Inference Systems", in *The 31st International Conference on Software Engineering and Knowledge Engineering*, Lisbon, 2019. DOI: 10.18293/SEKE2019-039.

[33] Ó. M. Pereira, D. D. Regateiro, and R. L. Aguiar, "Distributed And Typed Role-Based Access Control Mechanisms Driven By CRUD Expressions", *International Journal of Computer Science: Theory and Application*, vol. 2, no. 1, pp. 1–11, Oct. 2014, ISSN: 2336-0984. [Online]. Available: http://orb-academic.org/index.php/journal-of-computer-science/article/view/35.

[34] Ó. M. Pereira, D. D. Regateiro, and R. L. Aguiar, "Protecting Databases from Schema Disclosure - A CRUD-Based Protection Model", in *Proceedings of the 13th International Joint Conference on e-Business and Telecommunications*, SCITEPRESS - Science and Technology Publications, 2016, pp. 292–301, ISBN: 978-989-758-196-0. DOI: 10.5220/0005967402920301.

[35] D. D. Regateiro, Ó. M. Pereira, and R. L. Aguiar, "SPDC: Secure Proxied Database Connectivity", in *Proceedings of the 6th International Conference on Data Science, Technology and Applications*, Madrid, Spain: SCITEPRESS - Science and Technology Publications, 2017, pp. 56–66, ISBN: 978-989-758-255-4. DOI: 10.5220/0006424500560066.

[36] D. D. Regateiro, Ó. M. Pereira, and R. L. Aguiar, "Server-Side Database Credentials: A Security Enhancing Approach for Database Access", en, in *Communications in Computer and Information Science*, vol. 814, 2018, pp. 215–236, ISBN: 9783319948089. DOI: 10.1007/978-3-319-94809-6_11.

[37] D. D. Regateiro, Ó. M. Pereira, and R. L. Aguiar, "A Search Space Optimization Method For Fuzzy Access Control Auditing", *Knowledge and Information Systems*, 2019. DOI: 10.1007/s10115-020-01480-1.

[38] D. D. Regateiro, Ó. M. Pereira, and R. L. Aguiar, "Supporting Pre-shared Keys in Closed Implementations of TLS", in *Proceedings of the 6th International Conference on Data Science, Technology and Applications*, Madrid, Spain: SCITEPRESS - Science and Technology Publications, 2017, pp. 192–199, ISBN: 978-989-758-255-4. DOI: 10.5220/0006424701920199.

[39] Ó. M. Pereira *et al.*, "Mediator framework for inserting xDRs into Hadoop", in *2016 IEEE Symposium on Computers and Communication (ISCC)*, IEEE, Jun. 2016, pp. 547–554, ISBN: 978-1-5090-0679-3. DOI: 10.1109/ISCC.2016.7543795.

[40] G. Chen and T. T. Pham, *Introduction to Fuzzy Sets, Fuzzy Logic and Fuzzy Control Systems*. 2001, p. 329, ISBN: 0849316588.

[41] C. Martínez-García, G. Navarro-Arribas, and J. Borrell, "Fuzzy Role-Based Access Control", *Information Processing Letters*, vol. 111, no. 10, pp. 483–487, 2011, ISSN: 0020-0190. DOI: 10.1016/j.ipl.2011.02.010.

[42] A. Kayes *et al.*, "Context-aware access control with imprecise context characterization for cloud-based data resources", *Future Generation Computer Systems*, vol. 93, pp. 237–255, Apr. 2019, ISSN: 0167-739X. DOI: 10.1016/j.future.2018.10.036.

[43] J. Li, Y. Bai, and N. Zaman, "A Fuzzy Modeling Approach for Risk-Based Access Control in eHealth Cloud", in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, IEEE, Jul. 2013, pp. 17–23, ISBN: 978-0-7695-5022-0. DOI: 10.1109/TrustCom.2013.66.

[44] H. F. Atlam *et al.*, "Fuzzy Logic with Expert Judgment to Implement an Adaptive Risk-Based Access Control Model for IoT", *Mobile Networks and Applications*, Jan. 2019, ISSN: 1383-469X. DOI: 10.1007/s11036-019-01214-w.

[45] E. Mamdani and S. Assilian, "An experiment in linguistic synthesis with a fuzzy logic controller", *International Journal of Man-Machine Studies*, vol. 7, no. 1, pp. 1–13, 1975. DOI: 10.1016/S0020-7373(75)80002-2.

[46] M. Sugeno, "Industrial applications of fuzzy control", *Elsevier Science Pub. Co.*, 1985.

[47] Oracle, *The Java Persistence API - A Simpler Programming Model for Entity Persistence*, 2006. [Online]. Available: http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html (visited on 03/03/2014).

[48] Eclipse, *EclipseLink*, 2008. [Online]. Available: https://www.eclipse.org/eclipselink/ (visited on 04/28/2014).

[49] A. Chlipala, "Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications", *Proceedings of the Ninth USENIX Symposium on Operating Systems Design and Implementation*, pp. 105–118, 2010. [Online]. Available: http://www.usenix.org/events/osdi10/tech/full_papers/Chlipala.pdf.

[50] J. Abramov *et al.*, "A methodology for integrating access control policies within database development", *Computers & Security*, vol. 31, no. 3, pp. 299–314, May 2012, ISSN: 1674048. DOI: 10.1016/j.cose.2012.01.004.

[51] R. Agrawal *et al.*, "Hippocratic databases", *Proceedings of the 28th ...*, vol. 4, no. 1890, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=1287383.

[52] J. Padma and Y. Silva, "Hippocratic PostgreSQL", *International Conference on Data Engineering*, no. 25, 2009, ISSN: 1063-6382. DOI: 10.1109/ICDE.2009.126.

[53] K. LeFevre *et al.*, "Limiting disclosure in hippocratic databases", *Proceedings of the ...*, pp. 108–119, 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1316701.

[54] P. Ashley *et al.*, "W3C Enterprise privacy authorization language (EPAL 1.2)", Tech. Rep., 2003, pp. 1–58. [Online]. Available: https://www.w3.org/Submission/2003/SUBM-EPAL-20031110.

[55] L. Cranor *et al.*, "The Platform for Privacy Preferences 1.0 (P3P1.0) Specification", *W3C*, vol. 0, pp. 1–76, 2002. [Online]. Available: http://www.w3.org/TR/P3P/.

[56] G. Zhang and M. Parashar, "Dynamic context-aware access control for grid applications", *Proceedings First Latin American Web Congress*, 2003. DOI: 10.1109/GRID.2003.1261704.

[57] B. Corcoran, N. Swamy, and M. Hicks, "Cross-tier, label-based security enforcement for web applications", *ACM SIGMOD International Conference on Management of Data*, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1559875.

[58] E. Cooper *et al.*, "Links: Web Programming Without Tiers", *Formal Methods for Components and Objects*, 2007. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-74792-5_12.

[59] N. Swamy, B. J. Corcoran, and M. Hicks, "Fable: A Language for Enforcing User-defined Security Policies", in *IEEE Symposium on Security and Privacy*, IEEE, May 2008, pp. 369–383, ISBN: 978-0-7695-3168-7. DOI: 10.1109/SP.2008.29.

[60] Y. Zhu *et al.*, *JIF: Java + information flow*, 2012. [Online]. Available: http://www.cs.cornell.edu/jif/.

[61] L. E. Olson, C. a. Gunter, and P. Madhusudan, "A formal framework for reflective database access control policies", *Proceedings of the 15th ACM conference on Computer and communications security - CCS '08*, p. 289, 2008. DOI: 10.1145/1455770.1455808. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1455770.1455808.

[62] A. Bonner, "Transaction datalog: A compositional language for transaction programming", *Database Programming Languages*, pp. 1–30, 1997. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-64823-2_21.

[63] L. E. Olson *et al.*, "Implementing Reflective Access Control in SQL", *Data and Applications Security XXIII*, pp. 17–32, 2009. DOI: 10.1007/978-3-642-03007-9_2.

[64] B. Morin *et al.*, "Security-driven model-based dynamic adaptation", *Proceedings of the IEEE/ACM international conference on Automated software engineering*, p. 205, 2010. DOI: 10.1145/1858996.1859040.

[65] Oracle, *Java EE*. [Online]. Available: http://www.oracle.com/technetwork/java/javaee/overview/index.html.

[66] J. Fischer *et al.*, "Fine-Grained Access Control with Object-Sensitive Roles", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5653 LNCS, 2009, pp. 173–194, ISBN: 3642030122. DOI: 10.1007/978-3-642-03013-0_9.

[67] J. Zarnett, M. Tripunitara, and P. Lam, "Role-based access control (RBAC) in Java via proxy objects using annotations", *Proceeding of the 15th ACM symposium on Access control models and technologies - SACMAT '10*, p. 79, 2010. DOI: 10.1145/1809842.1809858.

[68] Oracle, *Java RMI*. [Online]. Available: http://docs.oracle.com/javase/tutorial/rmi/ (visited on 01/28/2014).

[69] S. Chaudhuri, T. Dutta, and S. Sudarshan, "Fine Grained Authorization Through Predicated Grants", in *Proceedings - International Conference on Data Engineering*, Istanbul, 2007, pp. 1174–1183, ISBN: 1424408032. DOI: 10.1109/ICDE.2007.368976.

[70] L. Caires *et al.*, "Type-Based Access Control in Data-Centric Systems", *Programming Languages and Systems*, pp. 136–155, 2011. DOI: 10.1007/978-3-642-19718-5_8.

[71] G. Ahn and H. Hu, "Towards realizing a formal RBAC model in real systems", *Proceedings of the 12th ACM symposium on Access control models and technologies*, p. 215, 2007. DOI: 10.1145/1266840.1266875.

[72] OASIS, *XACML 3.0*, 2010. [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml (visited on 05/11/2014).

[73] K. K. Jayaraman *et al.*, "MOHAWK: Abstraction-refinement and bound-estimation for verifying access control policies", *ACM Transactions on Information and System Security*, vol. 15, no. 4, pp. 1–28, 2013. DOI: 10.1145/2445566.2445570.

[74] D. S. Wallach, A. W. Appel, and E. W. Felten, "SAFKASI: a security mechanism for language-based systems", *ACM Transactions on Software Engineering and Methodology*, vol. 9, no. 212, pp. 341–378, 2000, ISSN: 1049331X. DOI: 10.1145/363516.363520.

[75] H. Singh *et al.*, "Real-Life Applications of Fuzzy Logic", *Advances in Fuzzy Systems*, vol. 2013, pp. 1–3, 2013, ISSN: 1687-7101. DOI: 10.1155/2013/581879.

[76] J. Hao *et al.*, "Fine-grained data access control with attribute-hiding policy for cloud-based IoT", *Computer Networks*, vol. 153, pp. 1–10, Apr. 2019, ISSN: 13891286. DOI: 10.1016/j.comnet.2019.02.008.

[77] H. Ouechtati, N. B. Azzouna, and L. B. Said, "A Fuzzy Logic Based Trust-ABAC Model for the Internet of Things", in *Advances in Intelligent Systems and Computing*, ser. Advances in Intelligent Systems and Computing, L. Barolli, M. Takizawa, F. Xhafa, and T. Enokido, Eds., vol. 926, Cham: Springer International Publishing, 2020, pp. 1157–1168, ISBN: 978-3-030-15031-0. DOI: 10.1007/978-3-030-15032-7_97.

[78] C. Janikow, "Fuzzy decision trees: Issues and methods", *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics)*, vol. 28, no. 1, pp. 1–14, 1998, ISSN: 1083-4419. DOI: 10.1109/3477.658573.

[79] S. A. Sert and A. Yazici, "Optimizing the Performance of Rule-Based Fuzzy Routing Algorithms in Wireless Sensor Networks", *IEEE International Conference on Fuzzy Systems*, vol. 2019-June, pp. 1–6, 2019, ISSN: 10987584. DOI: 10.1109/FUZZ-IEEE.2019.8858920.

[80] E. Erhieyovwe, P. Oghenekaro, and N. Oluwole, "An Object Relational Mapping Technique for Java Framework", *International Journal of Engineering Science Invention*, vol. 2, no. 6, pp. 1–9, 2013, ISSN: 2319-6726.

[81] C. Russell, "Bridging the Object-Relational Divide", *Queue*, vol. 6, no. June, p. 18, 2008, ISSN: 15427730. DOI: 10.1145/1394127.1394139.

[82] Eclipse, *Understanding EclipseLink 2.4*, June. 2013. [Online]. Available: https://www.eclipse.org/eclipselink/documentation/2.4/eclipselink_otlcg.pdf.

[83] S. Bagui, "Achievements and Weaknesses of Object-Oriented Databases", *Journal of Object Technology*, vol. 2, no. 4, pp. 29–41, 2003, ISSN: 16601769. DOI: 10.5381/jot.2003.2.4.c2.

[84] H. Garcia-Molina, J. D. Ullman, and J. Widom, "Stored Procedures", in *Database systems: the complete book*, 2nd E., 2008, ch. 9.4, pp. 391–404, ISBN: 978-0131873254.

[85] S. Sumathi and S. Esakkirajan, *Fundamentals of Relational Database Management Systems*. 2007, ISBN: 978-3642080128.

[86] S. Rohilla and P. K. Mittal, "Database Security by Preventing SQL Injection Attacks in Stored Procedures", *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, no. 11, pp. 915–919, 2013, ISSN: 2277-128X.

[87] A. Roichman and E. Gudes, "Fine-grained access control to web databases", *Proceedings of the 12th ACM symposium on Access control models and technologies - SACMAT '07*, p. 31, 2007. DOI: 10.1145/1266840.1266846.

[88] J. Wilson, "Views as the security objects in a multilevel secure relational database management system", *Proceedings. 1988 IEEE Symposium on Security and Privacy*, 1988. DOI: 10.1109/SECPRI.1988.8099.

[89] IETF, *RFC 5246: The Transport Layer Security (TLS) Protocol - Version 1.2*, 2008. [Online]. Available: http://tools.ietf.org/html/rfc5246.

[90] IETF, *RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3*, 2018. [Online]. Available: https://tools.ietf.org/html/rfc8446.

[91] IETF, *RFC 6101: The Secure Sockets Layer (SSL) Protocol Version 3.0*. [Online]. Available: http://tools.ietf.org/html/rfc6101.

[92] C. Adams and S. Lloyd, *Understanding public-key infrastructure: concepts, standards, and deployment considerations*. Sams Publishing, 1999.

[93] J. de Lavarene and J. de Lavarene, *SSL With Oracle JDBC Thin Driver*, 2010. [Online]. Available: http://www.oracle.com/technetwork/topics/wp-oracle-jdbc-thin-ssl-130128.pdf.

[94] S. Khandelwal, *Chinese Certificate Authority 'mistakenly' gave out SSL Certs for GitHub Domains*, 2016. [Online]. Available: https://thehackernews.com/2016/08/github-ssl-certificate.html.

[95] C. Ellison and B. Schneier, "Ten risks of PKI: What you are not being told about public key infrastructure", in *Public Key Infrastructure: Building Trusted Applications and Web Services*, 2004, ISBN: 9780203498156. DOI: 10.1201/9780203498156.

[96] A. Vishwakarma, "Virtual private networks", in *Network Security Attacks and Countermeasures*, 2016, ISBN: 9781466687622. DOI: 10.4018/978-1-4666-8761-5.ch003.

[97] IETF, *The Kerberos Network Authentication Service (V5)*, 2005. [Online]. Available: https://tools.ietf.org/html/rfc4120.

[98] IETF, *RFC 2865: Remote Authentication Dial In User Service (RADIUS)*, 2000. [Online]. Available: https://tools.ietf.org/html/rfc2865.

[99] Oracle, *Authentication Using Third-Party Services*. [Online]. Available: https://docs.oracle.com/cd/B19306_01/network.102/b14266/authmeth.htm#i1009853 (visited on 08/13/2016).

[100] R. Oppliger, R. Hauser, and D. Basin, "SSL/TLS session-aware user authentication - Or how to effectively thwart the man-in-the-middle", *Computer Communications*, vol. 29, no. 12, pp. 2238–2246, 2006, ISSN: 1403664. DOI: 10.1016/j.comcom.2006.03.004.

[101] R. Oppliger, R. Hauser, and D. Basin, "SSL/TLS session-aware user authentication revisited", *Computers and Security*, vol. 27, pp. 64–70, 2008, ISSN: 1674048. DOI: 10.1016/j.cose.2008.04.005.

[102] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. López, K. Papagiannaki, P. Rodriguez Rodriguez, and P. Steenkiste, "Multi-Context TLS (mcTLS)", *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 5, pp. 199–212, Aug. 2015, ISSN: 1464833. DOI: 10.1145/2829988.2787482.

[103] P. Ferraro, *HA-JDBC: High-Availability JDBC*. [Online]. Available: https://ha-jdbc.github.io (visited on 09/13/2016).

[104] M. Zimmerman, *Biometrics and User Authentication*, 2003. [Online]. Available: https://www.sans.org/reading-room/whitepapers/authentication/biometrics-user-authentication-122.

[105] G. Fawkes, *Report: Data Breach in Biometric Security Platform Affecting Millions of Users*, 2019. [Online]. Available: https://www.vpnmentor.com/blog/report-biostar2-leak/ (visited on 01/10/2020).

[106] M. H. Ibrahim, "REFLECTION IN OBJECT-ORIENTED PROGRAMMING", *International Journal on Artificial Intelligence Tools*, vol. 01, no. 01, pp. 117–136, Mar. 1992, ISSN: 0218-2130. DOI: 10.1142/S0218213092000156.

[107] Oracle, *The Reflection API*. [Online]. Available: http://docs.oracle.com/javase/tutorial/reflect/ (visited on 01/06/2014).

[108] R. Shay *et al.*, "Designing password policies for strength and usability", *ACM Transactions on Information and System Security*, vol. 18, no. 4, pp. 1–34, May 2016, ISSN: 10949224. DOI: 10.1145/2891411.

[109] Oracle, *RMI - Remote Method Invocation*. [Online]. Available: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html.

[110] G. Canfora, C. Visaggio, and V. Paradiso, "A Test Framework for Assessing Effectiveness of the Data Privacy Policy's Implementation into Relational Databases", *2009 International Conference on Availability, Reliability and Security*, pp. 240–247, 2009. DOI: 10.1109/ARES.2009.153.

[111] P. Eronen and H. Tschofenig, "[PSK] Pre-Shared Key Ciphersuites for Transport Layer Security (TLS) [RFC 4279]", *RFC 4279*, pp. 1–15, 2005.

[112] W. Diffie and M. Hellman, "New directions in cryptography", *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 29–40, 1976, ISSN: 0018-9448. DOI: 10.1109/TIT.1976.1055638.

[113] S. Marechal, *Advances in password cracking*, 1. 2008, vol. 4, pp. 73–81. DOI: 10.1007/s11416-007-0064-y.

[114] IEC, *Fuzzy Control Programming (IEC 1131-7 CD1)*, 1997. [Online]. Available: http://www.fuzzytech.com/binaries/ieccd1.pdf.

[115] V. C. V. Hu, D. F. Ferraiolo, and D. R. Kuhn, "Assessment of access control systems", *Nistir 7316*, p. 60, 2006. DOI: 10.6028/NIST.IR.7316.

[116] C. Günther, "An identity-based key-exchange protocol", *Advances in Cryptology - Eurocrypt'89*, vol. 434, pp. 29–37, 1989. DOI: 10.1007/3-540-46885-4_5.

[117] J. Nielsen, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, ISBN: 0125184069.

[118] P. Cingolani and J. Alcalá-Fdez, "jFuzzyLogic: a robust and flexible Fuzzy-Logic inference system language implementation", in *2012 IEEE International Conference on Fuzzy Systems*, IEEE, Jun. 2012, pp. 1–8, ISBN: 978-1-4673-1506-7. DOI: 10.1109/FUZZ-IEEE.2012.6251215.

[119] P. Cingolani and J. Alcalá-Fdez, "jFuzzyLogic: a Java Library to Design Fuzzy Logic Controllers According to the Standard for Fuzzy Control Programming", *International Journal of Computational Intelligence Systems*, vol. 6, no. sup1, pp. 61–75, Jun. 2013, ISSN: 1875-6891. DOI: 10.1080/18756891.2013.818190.