



**Diogo José
Domingues Regateiro**

**RBAC seguro, dinâmico e distribuído para
aplicações relacionais**

**A secure, distributed and dynamic RBAC for
relational applications**



**Diogo José
Domingues Regateiro**

**RBAC seguro, dinâmico e distribuído para
aplicações relacionais**

**A secure, distributed and dynamic RBAC for
relational applications**

Tese apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Óscar 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.

Dedico este trabalho aos meus pais pelo seu incansável apoio.

o júri / the jury

presidente / president

Prof. Doutor André Ventura da Cruz Marnoto Zúquete
professor auxiliar da Universidade de Aveiro

vogais / examiners committee

Prof. Doutor Gabriel de Sousa Torcato David
professor associado da Faculdade de Engenharia da Universidade do Porto

Prof. Doutor Óscar Narciso Mortágua Pereira
professor auxiliar da Universidade de Aveiro

agradecimentos

Aproveito agora para agradecer a todas as pessoas que me ajudaram e apoiaram durante o meu percurso académico e na realização deste trabalho.

Ao professor Doutor Óscar Mortágua Pereira, meu orientador, por me ter ajudado sempre que precisei durante este trabalho. Ao professor Doutor Rui Luís Aguiar, meu co-orientador, pela sua visão crítica e pelo apoio que me proporcionou em certos pontos do trabalho, e ao professor Doutor André Zúquete, pela sua ajuda em parte deste trabalho.

Agradeço também à minha família pelo apoio e incentivo que me deram e aos meus vários colegas de curso, especialmente ao Engenheiro Tiago Magalhães pelo seu sentido de responsabilidade e ajuda nos diversos momentos do meu percurso académico.

palavras-chave

segurança informática, controlo de acesso, RBAC, arquitectura de software, engenharia de software, sistemas distribuídos, middleware, bases de dados relacionais, engenharia de software baseada em componentes, engenharia de software automatizada

resumo

Atualmente, aplicações que acedem a bases de dados utilizam ferramentas como o Java Database Connectivity, Hibernate ou ADO.NET para aceder aos dados nelas armazenados. Estas ferramentas estão desenhadas para unir os paradigmas das bases de dados relacionais e da programação orientada a objetos, mas não estão preocupados com as políticas de controlo de acesso a aplicar. Portanto, os programadores de aplicações têm de dominar as políticas estabelecidas a fim de desenvolver aplicações em conformidade com as políticas de controlo de acesso estabelecidas.. Além disso, existem situações em que as políticas de controlo de acesso podem evoluir dinamicamente. Nestes casos, torna-se difícil adequar os mecanismos de controlo de acesso. Este desafio motivou o desenvolvimento de uma extensão ao modelo de controlo de acesso baseado em papéis (RBAC) que define como permissões sequências de expressões para criar, ler, atualizar e apagar (CRUD) informação e as interfaces de acesso a cada uma delas. A partir destas permissões podem ser gerados artefactos de segurança do lado dos clientes, i.e. de uma forma distribuída, que lhes permitem aceder à informação armazenada na base de dados segundo as políticas definidas. Por cima desta extensão também foi criada uma camada de segurança para tornar o controlo de acesso seguro e obrigatório. Para a extensão do modelo RBAC este trabalho baseou-se num trabalho anterior que criou uma arquitectura dinâmica de controlo de acesso para aplicações de bases de dados relacionais, aqui referida como DACA (Dynamic Access Control Architecture). DACA utiliza informação da lógica de negócio e as políticas de controlo de acesso que foram definidos para criar dinamicamente os artefactos de segurança para as aplicações. Em situações onde as políticas de controle de acesso evoluem de forma dinâmica, os artefactos de segurança são ajustados automaticamente. Este trabalho base, no entanto, define como permissões as expressões CRUD, podendo estas ser executadas em qualquer ordem, e necessita de uma camada de segurança adequada para autenticar utilizadores e proteger os dados sensíveis de intrusos. Portanto, neste trabalho, pretende-se criar uma nova arquitectura, chamada “S-DRACA” (Secure, Dynamic and Distributed Role-based Access Control Architecture), que estende o trabalho feito no âmbito do DACA para que este seja capaz de garantir que sejam cumpridas sequência de expressões CRUD que as aplicações podem executar e que estão associados aos seus papéis nas políticas RBAC e desenvolver uma camada de segurança adequada para a tornar segura. Discutimos, também, o seu desempenho e aplicabilidade em outros ambientes sem ser em bases de dados relacionais.

keywords

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

abstract

Nowadays, database application use tools like Java Database Connectivity, Hibernate or ADO.NET to access data stored in databases. These tools are designed to bring together the relational database and object-oriented programming paradigms, forsaking applied access control policies. Hence, the application developers must master the established policies as a means to develop software that is conformant with the established access control policies. Furthermore, there are situations where these policies can evolve dynamically. In these cases it becomes hard to adjust the access control mechanisms. This challenge has led to the development of an extension to the role based access control (RBAC) model where permissions are defined as a sequence of create, read, update and delete (CRUD) expressions that can be executed and the interfaces to access them. From these permissions it's possible to generate security artefacts on the client side, i.e. in a distributed manner, which allows the clients to access the stored data while satisfying the security policies defined. On top of this model extension, a security layer has also been created in order to make the access control secure and obligatory. For the RBAC model extension this work leverages a previous work that created a dynamic access control architecture for relational applications, here referred to as DACA (Dynamic Access Control Architecture). DACA uses business logic information and the defined access control policies to build dynamically the security artefacts for the applications. In situations where the access control policies can evolve dynamically, the security artefacts are adjusted automatically. This base work, however, defines as permissions CRUD expressions, which can be executed in any order, and needs an adequate security layer to authenticate users and protect the system from intruders. Hence, this work aims to create a new architecture, called "S-DRACA" (Secure, Dynamic and Distributed Role-based Access Control Architecture), which extends the work done with DACA so that it is capable of enforcing sequences of CRUD expressions that the applications can execute if the sequences are associated with their roles and the development of a security layer to make it secure. We discuss as well the performance of this system and its applicability to other environments outside of relational databases.

Table of Contents

List of Figures	IV
List of Tables.....	V
List of Acronyms	VI
1 Introduction	1
1.1 Problem formulation	1
1.2 Proposed solution	2
1.3 Contributions.....	3
1.4 Tools and infrastructures used	3
1.5 Structure of the dissertation.....	4
2 State of the art and related work.....	5
2.1 Access Control Policies	5
2.1.1 Discretionary Access Control	6
2.1.2 Mandatory Access Control.....	7
2.1.3 Role-Based Access Control.....	9
2.1.4 Attribute-Based Access Control	12
2.2 Relational database management systems	13
2.2.1 Relational model	13
2.2.2 Policy control enforcement mechanisms	15
2.2.3 Application's database access libraries	17
2.2.3.1 Java Database Connectivity	17
2.2.3.2 ADO.NET	19
2.2.3.3 Hibernate.....	19
2.2.3.4 LINQ.....	19
2.3 Related work.....	20
2.4 Summary.....	26
3 Technological Background.....	27
3.1 Service coordination.....	27
3.1.1 Orchestration	27
3.1.2 Choreography.....	28
3.2 Java	29
3.2.1 Reflection	29
3.2.2 Annotations.....	30
3.3 Cryptography and authentication.....	32
3.3.1 Hash functions.....	32
3.3.2 Encryption	33
3.3.2.1 Keys.....	34
3.3.2.1.1 Symmetric Keys.....	34
3.3.2.1.2 Diffie-Hellman Key Exchange	35
3.3.2.1.3 Asymmetric Keys	36

3.3.2.1.4	Public Key Certificates.....	37
3.3.2.2	SSL/TLS.....	38
3.3.2.2.1	Basic protocol.....	38
3.3.2.2.2	Anonymous SSL/TLS	40
3.3.3	Authentication	40
3.3.3.1	Password Authentication Protocol.....	41
3.3.3.2	Challenge-Response Authentication Mechanism.....	41
3.3.3.3	Encrypted Key Exchange	42
3.3.3.4	Smart card.....	43
3.3.4	Security attacks	44
3.3.4.1	Replay attack.....	45
3.3.4.2	Exhaustive key search.....	45
3.3.4.3	Dictionary attack	45
3.3.4.4	Pre-computed dictionary attack.....	46
3.3.4.5	Reflection attack.....	46
3.3.4.6	Man-in-the-middle attack.....	47
3.4	Summary.....	48
4	Secure, dynamic and distributed RBAC Architecture.....	49
4.1	S-DRACA overall architecture.....	49
4.1.1	DACA overview.....	50
4.1.2	S-DRACA overview	51
4.1.3	S-DRACA's architecture.....	53
4.1.4	Static implementation of the RBAC policies.....	54
4.1.5	Summary	56
4.2	Access control policies layer	57
4.2.1	Conceptual RBAC policy extension	57
4.2.2	RBAC model extension.....	58
4.2.3	Software architectural model	61
4.2.3.1	Direct access mode interface	62
4.2.3.2	Indirect access mode interface	62
4.2.4	Summary	63
4.3	Sequence controller component	65
4.3.1	Conceptual model.....	65
4.3.2	Implementation solutions.....	66
4.3.2.1	Using the role's security data structures interface with enumerated classes	66
4.3.2.2	Using the role's security data structures interface with 'next' references	68
4.3.2.3	Using the Business Schemas	69
4.3.2.4	Common problems and the chosen solution.....	72
4.3.3	Implementation of the sequence controller	73
4.3.3.1	Initialization.....	75
4.3.3.2	Sequence control lifecycle.....	75
4.3.4	Summary	78
4.4	Security layer	79
4.4.1	Authentication and data encryption	80
4.4.1.1	Hash-based Password Authentication Protocol.....	81
4.4.1.2	Challenge-Response based protocol.....	81
4.4.1.3	SSL/TLS.....	82
4.4.1.3.1	Server Certificate.....	82

4.4.1.3.2	Pre-Shared Key + Anonymous Diffie-Hellman	83
4.4.1.3.3	Simple Password Exponential Key Exchange	85
4.4.1.3.4	Secure Remote Password	85
4.4.1.4	Two Factor Authentication	86
4.4.2	Query protection	86
4.4.3	Secure data store communication	89
4.4.4	Summary	90
4.5	Performance	91
4.5.1	Environment	91
4.5.2	Initialization tests	92
4.5.3	Execution tests	94
4.5.4	Results discussion	96
4.5.5	Summary	97
4.6	Proof of Concept	99
4.6.1	Policy Extractor	99
4.6.2	Sample applications	102
4.6.3	Summary	107
5	Conclusion	109
5.1	Sequence controller	109
5.2	Security layer	109
5.3	Discussion	110
5.3.1	Application to other access control policies	111
5.3.2	Integration with other data sources	111
5.3.2.1	Integration with Hive	113
5.3.2.2	The insertion conflict	114
5.3.3	Future work	114
6	References	117
Appendix A – SSL/TLS and authentication implementation details		i
A.1	Hash-based password authentication protocol	i
A.2	Challenge-response authentication mechanism	i
A.3	Two factor authentication	ii
A.4	SSL/TLS with certificates	iii
Appendix B – CoherentPaaS project and its applicability		v
B.1	Overview	v
B.2	Applicability	v

List of Figures

Figure 1. Role relationships.	9
Figure 2. RBAC96's proposed RBAC ₃ model.	10
Figure 3. Basic ABAC model.	12
Figure 4. PDP-PEP security layer model.....	15
Figure 5. SAAM policy enforcement architecture.	16
Figure 6. JDBC architecture.....	17
Figure 7. JDBC statement usage example.	18
Figure 8. JDBC prepared statement usage example.	18
Figure 9. JDBC callable statement usage example.	18
Figure 10. Result set row iteration.	18
Figure 11. Service orchestration.	28
Figure 12. Service choreography.	28
Figure 13. Instantiation and invocation using reflection.	30
Figure 14. JPA data class example.	31
Figure 15. Example of a hypothetical hash function.	33
Figure 16. Basic usage of encryption to send data over an unsecure channel.....	33
Figure 17. Message encryption over an unsecure channel using a shared symmetric key.	34
Figure 18. Message encryption over an unsecure network using an asymmetric key pair.	36
Figure 19. Message signing using an asymmetric key pair.	37
Figure 20. Basic TLS protocol using server certificate.	39
Figure 21. MITM attack exemplification where Eve is the attacker.	47
Figure 22. S-DRACA stack.....	50
Figure 23. DACA architecture block diagram.	51
Figure 24. S-DRACA architecture block diagram.	52
Figure 25. The S-DRACA architecture.	54
Figure 26. GetBus communication process.....	55
Figure 27. Sample code from the SQL Server generator.....	56
Figure 28. Example of a digraph.	60
Figure 29. Examples of valid paths derived from Figure 28.	60
Figure 30. Extension for the RBAC model.....	60
Figure 31. The implemented RBAC model.....	61
Figure 32. Software architectural model for one role.....	62
Figure 33. Automated building process of Business Schemas.	63
Figure 34. Sequence controller conceptual model.	66
Figure 35. Security data structures using enumerated classes.....	67
Figure 36. Example of the usage of Business Schemas with sequence control using enumerated classes where customers are selected and a new order inserted.	67
Figure 37. Security data structures with next references.	68
Figure 38. Example of the usage of Business Schemas with sequence control using next references where customers are selected and a new order inserted.....	69
Figure 39. Security data structure comprising information about <i>Role_B1</i>	70
Figure 40. Interaction between application and the Factory class.	70
Figure 41. The Factory requests an instance for a Business Schema.	70
Figure 42. The S_Orders Business Schema provides an edge method to step forward to next Business Schema.....	71
Figure 43. Validation process for the instantiation process of Business Schemas.....	71
Figure 44. Using the Business Schema's to get the next one without aliases.	72
Figure 45. Using aliases in place of Business Schemas for sequence 1.	73
Figure 46. Using aliases in place of Business Schemas for sequence 2.	73
Figure 47. Implementation of the sequence controller component in the Business Manager.	74

Figure 48. Software model for the sequence control mechanism.....	75
Figure 49. Sequence diagram of the sequence control mechanism for requesting the first Business Schema and making requests.	76
Figure 50. Sequence diagram of the sequence control mechanism for requesting the next Business Schema.	77
Figure 51. The abstract DacClientAuthenticator class.	80
Figure 52. Server socket creation process with the password-authenticated key exchange.	84
Figure 53. Client socket creation with the password-authenticated key exchange.....	84
Figure 54. Signature of the RemoteCall stored procedure.	87
Figure 55. Graph of the initialization times in Table 14, Table 15 and Table 16.	93
Figure 56. Graph of the execution times in Table 17 for the single select scenario.	95
Figure 57. Graph of the execution times in Table 17 for the two selects scenario.	95
Figure 58. Graph of the execution times in Table 17 for the SISUS scenario.	96
Figure 59. Proof of concept architectural colution.	99
Figure 60. Policy Extractor annotation.	100
Figure 61. Annotation processor declaration.....	100
Figure 62. Annotation processor's process method signature.	100
Figure 63. The Business Schemas interfaces generation process.....	101
Figure 64. InterfaceGenerator main method.	101
Figure 65. Generated sources folder.	102
Figure 66. Initial window of the configurator.....	102
Figure 67. Client application showing the list of customers.	103
Figure 68. Code that adds information into the Customers' table.	103
Figure 69. Second step in updating an order.	104
Figure 70. The final update step and confirmation dialog.	104
Figure 71. Configurator revocation list window, after revoking the first Business Schema in the second position.	105
Figure 72. Client application's error message for trying to execute a revoked Business Schema.	105
Figure 73. The client's unrestricted window.	105
Figure 74. The Configurator's user roles windows.	106
Figure 75. Client's unrestricted window after the user had its roles revoked.	106
Figure 76. The change listener interface.	106
Figure 77. Operation performed when policies change.....	106
Figure 78. New Business Schema generator model.	112

List of Tables

Table 1. Access control matrix example.	6
Table 2. Authorization table example equivalent to the access matrix control in Table 1.	7
Table 3. Example relation in a RDBMS.	14
Table 4. Diffie-Hellman protocol original implementation using encryption mathematics.....	35
Table 5. Password authentication protocol.	41
Table 6. Challenge-Response authentication mechanism protocol.	42
Table 7. MITM attack scenario.	47
Table 8. Roles, Business Schemas and CRUD expressions.	78
Table 9. Roles, sequences, revocation and crud lists.	78
Table 10. Example of two CRUD expressions in the Queries table.	88
Table 11. Example of information in the Operands table for the Queries in Table 10.	88
Table 12. Example of the generated identifiers for a client session.....	88
Table 13. Testing machine specification.....	92

Table 14. Initialization times for JDBC and S-DRACA when both relaying and TFA were used.....	92
Table 15. Initialization times for JDBC and S-DRACA when neither relaying nor TFA were used.	93
Table 16. Initialization times for JDBC and S-DRACA when relaying was used but not TFA.	93
Table 17. Performance execution times.....	94
Table 18. Hash-based Password Authentication Protocol messages.	i
Table 19. CRAM based authentication messages.	ii
Table 20. Two factor authentication protocol.	iii
Table 21. SSL/TLS with certificates upgrading protocol.	iv

List of Acronyms

ABAC	Attribute-Based Access Control
ACL	Access Control List
CC	Portuguese Citizen Card
CLI	Call Level Interfaces
DAC	Discretionary Access Control
DACA	Dynamic Access Control Architecture
DBMS	Database Management System
EKE	Encrypted Key Exchange
HDFS	Hadoop Distributed File System
IDAC	Direct Access Mode Interface
IDE	Integrated Development Environment
IIAM	Indirect Access Mode Interface
JDBC	Java Database Connectivity
JVM	Java Virtual Machine
LDS	Local Data Set
LINQ	Language Integrated Query
MAC	Mandatory Access Control / Message Authentication Code
OCSP	Online Certificate Status Protocol
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PSK	Pre-Shared Key
RBAC	Role Based Access Control
RDBMS	Relational Database Management System
SAAM	Secondary and Approximate Authorization Model
SDP	Secondary Decision Point
S-DRACA	Secure, Dynamic and Distributed Role-based Access Control Architecture
SPEKE	Simple Password Exponential Key Exchange
SQL	Structured Query Language
SRP	Secure Remote Password
SSL	Secure Sockets Layer
TFA	Two Factor Authentication
TLS	Transport Layer Security

1 Introduction

The constant need to store data has been regarded as a crucial part of any area of human development (Sumathi, 2007), which led to the storage of information in computer systems. However, there is a need to organize these information and/or data in a way that it could be easily accessed. The result is what we refer here as “databases”. Databases can be managed by database management systems (DBMS), where the data is traditionally accessed using a standard querying language. Securing sensitive data from unauthorized users is a major security concern and is a task to be accomplished in the present scenario.

A well-established solution restricts the access to sensitive data to the authorized users only, which is known as access control. This raised security concerns when the information is sensitive, especially when allowing only the authorized users to access said information. Existing database management solutions, address this concern by creating a security layer separated from the data layer, where access control mechanisms such as Role-based access control (RBAC) need to be implemented. The separation of these layers has implications when software solutions that provide access to these data are used by applications. Some examples of such software solutions are Hibernate (JBoss, 2001), JDBC (Oracle, 1997a), LINQ (Microsoft, 2007), ADO.NET (Microsoft, n.d.-a) etc. These software solutions are usually developed having compatibility with many DBMSs in mind and lacked over the access control policies concerns. Hence the applications that use these software solutions cannot apply these policies automatically. This means that the applications can unknowingly perform an unauthorized operation on the sensitive data through one of these software solutions, which raises runtime errors on each attempt. This led to the development of the Dynamic Access Control Architecture (DACA), an architecture that takes into account, the defined RBAC policies to generate security entities that can be used by the applications. If an operation is not authorized, the application will not be able to execute it via the generated entities. An operation in DACA is a create, read, update or delete (CRUD) expression that allows a user to read or modify the sensitive data. The main focus of this dissertation is to extend DACA to provide a new architecture solution, named “S-DRACA”, with a more refined control over what the users can do and to address some security vulnerabilities that affects it. In addition, we further aimed at evaluating its performance.

This section is divided as follows: Section 1.1 will present the problem addressed with this work, section 1.2 proposes the solution developed, section 1.3 presents the contributions made with this work, section 1.4 presents the tools and infrastructures used and section 1.5 explains the structure of the dissertation.

1.1 Problem formulation

In this section we will formulate the problem we are facing. DACA was able to address the lack of integration of RBAC policies in the development of applications that use relational databases as a data source, i.e. the problem where programmers could write and execute any CRUD expression without having any information provided by the tools such as JDBC regarding the database schema or the

access control policies. If this information is not provided by these tools then programmers are required to have a prior comprehensive knowledge of it. DACA was successful in solving this problem by generating security entities that the client applications can use from an extended RBAC model that associates CRUD expressions to roles and defines the authorized operations. These security entities provide a standard call level interface (CLI) for the applications to use CRUD expressions to access sensitive data. This CLI allows the applications to access the data through an interface similar to the one provided by JDBC. Additionally, it only provides the methods that are authorized in the access control policy. Since the security entities are generated and implemented at runtime, they can be adjusted to reflect the access policies changes.

However, with DACA the applications are allowed to execute any CRUD expression in any order they want. The only restriction is that the CRUD expression must be authorized for the role played by the user. This can sometimes lead to unintended data disclosure even when the user only uses authorized operations. In (Canfora, 2009) is shown that this problem exists even outside of DACA, so RBAC policies are not able to prevent this type of exploitation from the start unless the authorized CRUD expressions are designed carefully.

Another problem with DACA is that its implementation was focused on supporting the dynamic modification of the defined access control policies. Thus the authentication of users was not made secure in the sense that anyone listening to the communication between the client applications and the server, where the database is hosted, could easily impersonate the user by stealing his credentials. Furthermore, the sensitive data was not encrypted which could lead to sensitive information leaking and the fact that the generated entities used by the applications are in the client side made them vulnerable to user manipulation.

1.2 Proposed solution

In this section we will introduce the Secure, Dynamic and Distributed Role-based Access Control Architecture (S-DRACA), a solution that aims to solve the stated problems. S-DRACA uses an extension of the current RBAC model in order to restrict the user's ability to execute even authorized CRUD expressions and uses DACA as a base project. The RBAC model extension will allow a security expert to define the sequences of CRUD expressions to be authorized and what happens during their lifecycle, i.e. when the client application moves between CRUD expressions in a pre-defined sequence. Each sequence is meant to allow a user to perform some high level operation. By requiring the users to follow specific sequences it becomes easier to prevent unintended data disclosure.

To address the security problems raised in the last section we propose to integrate a security layer in S-DRACA. This layer will authenticate users by using several different authentication mechanisms, each with different security/communication overhead trade-offs. This layer will also encrypt data by using the SSL/TLS protocol and address some of the vulnerabilities generated by enforcing the access policies on the client side.

Therefore, S-DRACA is a dynamic access control architecture that still allows security experts to make changes to the access policies at any time with the knowledge that the access control mechanisms in the client applications will be adjusted accordingly. Now as compared to DACA, this provides the means to restrict the client applications to follow the pre-defined sequences of CRUD expressions. Furthermore, it also authenticates and transmits data more securely and many of the vulnerabilities that affected the system are also addressed.

1.3 Contributions

In this section we will introduce the contributions in the area achieved with S-DRACA. S-DRACA is an architecture with significance since it enforces access control policies that can change dynamically with the entities it generates. Furthermore, the RBAC model extension, allows the definition of sequences of authorized CRUD expressions. This is significant since it helps preventing unintended disclosure of information by using together authorized operations in an unforeseen way. These facts led to two publications. The paper “Role-Based Access Control Mechanisms”(Ó. M. Pereira, 2014a), which presented the new architecture for data access to relational databases and dynamic access control policies enforcement. It was submitted and accepted in the 19th IEEE symposium on computers and communications (ISCC). And a second paper, entitled “Extending RBAC Model to Control Sequences of CRUD Expressions”(Ó. M. Pereira, 2014b), presented the extension made to the RBAC model to support the definition and enforcement of sequences of CRUD expressions. It was submitted and accepted in the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE).

Another contribution achieved with S-DRACA is the security layer that was developed. Its significance resides in its ability to use a secure communication channel, established during the client authentication, to allow the client to communicate with the relational database management system (RDBMS) and in pushing the CRUD expressions from the client to the server. This contribution will have a paper that is yet to be defined.

1.4 Tools and infrastructures used

In this section we will present the tools and infrastructures used during the development of S-DRACA. Many tools used for the development of DACA were reused for the development of S-DRACA. The main programming language used is the version 8 of Java(Oracle, n.d.-a), due to the portability it grants to the applications written in it. To develop the new functionalities the Netbeans(Oracle, n.d.-b) integrated development environment (IDE) was used. This IDE organized the different components of S-DRACA and eased the development process. The DBMS used to host the database with the sensitive data and the access control policies was SQL Server 2012(Microsoft, 2012), hosted on a Windows 7 machine. To ease the manipulation of the databases the SQL Server Management Studio 2012 was used, which allowed the execution of queries and the manipulation of the data stored through a graphical user interface.

1.5 Structure of the dissertation

In this section we will present the structure of this dissertation. The dissertation is divided in the following main sections: section 2, where the state of the art is discussed, which includes some discussion of the access control policies, RDBMS and the related work; section 3 presents some technological background of technologies used in S-DRACA; section 4 introduces an overview of DACA and presents the work done to develop S-DRACA as well as a proof of concept; section 5 draws the conclusions of the development of S-DRACA, discusses the future work and the applicability of S-DRACA in other contexts; Appendix A presents the implementation details on some standard security components; Appendix B presents a European project where we are involved and how it can be used in the context of this work; and finally the last section contains the references used.

2 State of the art and related work

In this section we will start by presenting the most common access control policies, the RDBMSs and how the two are implemented together. Then we will discuss related work and how S-DRACA complements or expands on them.

This section is organized as follows. Section 2.1 introduces the main access control policies. Section 2.2 presents the RDBMSs, how the access policies can be enforced and the tools used by applications to build their data access layer. Section 2.3 presents the related work and section 2.4 will provide a brief summary of the presented material.

2.1 Access Control Policies

In this section we will introduce the current access control policies. Access control policies are one of many key aspects in order to obtain security in any technological system. The most important requirements that must be met are(Samarati, 2001): integrity, availability and secrecy. Integrity means that data and resources must be protected from non-authorized modifications. Availability means that data and resources must be always available to authorized users. Lastly, secrecy means that data and resources must be protected against non-authorized disclosure.

To protect the data and resources from non-authorized access a security layer is used that denies or allows access to them, which we will call access control. This access control layer can be implemented at different levels and places, such as at the client side or at the server side. This is further discussed in section 2.2.2.

The development of such an access control mechanism normally follows three distinct phases(Samarati et al., 2001): the definition of the access control policies, the creation of the security model to be followed and the definition of the access control enforcement mechanisms.

In the definition of the access control policies phase the rules are defined that specify exactly which entities have access to what resources. These are associated with the business logic (e.g. a receptionist should not have access to a patient's more personal information, but a doctor should). Because of this association the rules are dynamic and can change when the business logic changes.

The creation of the security model phase is where the access control policies are formally defined and are developed to show the security properties.

Finally, in the definition of the access control enforcement mechanisms phase the mechanisms responsible to actually enforce the policies defined in the previous phases. These mechanisms work as a middleware that intercepts every access attempt to the system. Every access attempt must be intercepted in order to enforce the defined rules and it must not be possible to change them in any way. They must also be confined to a small part of the system and must be small in order to be rigorously verifiable.

These phases allow for the separation of the policies and the mechanisms that enforce them. This separation has many advantages, e.g. the policies become independent from the mechanisms that enforce them, which allow us to analyze and discuss the policies without having to worry about the enforcement.

The access control policies can be placed in many different classes, of which we present four: the mandatory, the discretionary, the role-based and the attribute-based. The mandatory access control (MAC) policies are the ones that are set by a security policy administrator and the users cannot override them in any way. The discretionary access control (DAC) policies are the ones that are used to control the user's access taking into consideration their identity and/or groups they belong to. The controls are discretionary in the sense that a user with the right permission can grant permissions to other users. The role-based access control (RBAC) policies are the ones that are defined around a limited set of roles. Users can then be put into one or more roles for that system and the permissions they have are passed through the roles they have. Finally, the attribute-based access control (ABAC) policies are the ones that are evaluated against the attributes of entities (users and objects), actions and the environment that is relevant to the request. These four classes of access control policies are further discussed in the next sections.

This section is divided as follows: Section 2.1.1 will present DAC, section 2.1.2 will MAC, section 2.1.3 will present RBAC, and section 2.1.4 will present ABAC.

2.1.1 Discretionary Access Control

In this section we will present DAC. In computer security, DAC is an access control type defined as a means to restrict users' access to objects based on its identity and/or the groups they belong to, by explicitly defining what they can do to each resource (Vimercati, 2007). The controls are discretionary in the sense that a user with the permission to do so, is capable of delegating that permission (perhaps indirectly) on to any other user (unless restrained by some other access control mechanism), while the granting and revocation of permissions are regulated by an administrative policy (Samarati et al., 2001). In the operating system's context, the first proposal for the protection of resources was an access control matrix (Lampson, 1974) that is indexed by the users in one dimension and the resources in the other, where each cell indicates the permission a user has for a resource. Table 1 shows an example of one such matrix, from where we can notice that users can have read and/or write permissions on files and execute permissions if the resource is an executable.

Table 1. Access control matrix example.

User	File X	File Y	File Z	Executable W
A	Read/Write	-	-	Read/Execute
B	Read	-	Read/Write	-
C	-	Read	-	-

Changes to the matrix are performed using primitive operations, such as inserting/removing users and resources, or directly modifying the permissions a user has for a resource. One problem with

this approach is that the matrix is usually very sparse because normally only the creator of the resource has the permission to read and modify it, meaning the all the other users have no permissions unless the owner grants them, which translates into empty cells in the matrix. To prevent the waste of disk space there are three other mechanisms proposed in(Vimercati et al., 2007): authorization tables, access control lists (ACLs) and capabilities lists.

An authorization table consists of a table with three columns: one for the user, another for the permission and the third one for the resource itself. Table 2 shows an equivalent authorization table for the access control matrix in Table 1.

Table 2. Authorization table example equivalent to the access matrix control in Table 1.

User	Permission	Resource
A	Read	File X
A	Write	File X
A	Read	Executable W
A	Execute	Executable W
B	Read	File X
B	Read	File Z
B	Write	File Z
C	Read	File Y

In the case of ACLs each object is associated with a list of users and the permissions he possesses. Capabilities lists can be seen as inverted ACLs in the sense that now it's the users that have a list of resources associated with them. The list of resources contains, for each resource, the permission that user possesses.

2.1.2 Mandatory Access Control

In this section we will present MAC. MAC refers to a type of access control by which a central entity constrains the ability of a subject or initiator to access or generally perform some sort of operation on an object or target based on its regulations(Samarati et al., 2001). MAC distinguishes between users and subjects, where users are humans that use the system and the subjects are processes created by the users. This fact allows the system to control the indirect access to information via leaks or modification resultant from the execution of processes. Any operation by any subject on any object will be tested against the set of authorization rules (i.e. the policy) to determine if the operation is allowed. A DBMS, in its access control mechanism, can also enforce MAC. In this case the objects are tables, views, procedures, etc.

With MAC, users do not have the ability to override the policy and, for example, grant access to files that would otherwise be restricted. MAC-enabled systems allow policy administrators to implement organization-wide security policies. This allows security administrators to define a central policy that is guaranteed (in principle) to be enforced for all users.

The policies, when they are multilevel, are usually classified based on the associations between subjects and objects, named access class. An access class is one element from a partially ordered set of

classes, a set that is ordered by a dominance relationship. The set of classes can be any set of labels that has a dominance relationship defined on them. Most commonly, an access class is defined consisting of two components: a security level and a set of categories. The security level is an element from a hierarchically ordered set, such as Top Secret (TS), Secret (S), Confidential (C) and Unclassified (U), where $TS > S > C > U$. The set of categories is a subset of an unordered set, whose members symbolize areas of competence or functionality, e.g. NATO, Nuclear, Security, Research, etc.

The dominance relationship is defined as follows: an access class C_1 dominates another access class C_2 if and only if the security level of C_1 is greater than or equal to the security level of C_2 and the categories of C_1 include those present for C_2 . The mandatory policies can be classified having secrecy or integrity in mind.

The mandatory policies based on secrecy are based on the proposed models in (Bell, 1973). The main goal is to protect the information from unauthorized disclosure. The security level associated with a user is called a clearance and it reflects the level of trust the user has regarding the non-disclosure of the sensitive information to users not cleared to see it. The categories are used to provide a finer control that was not possible only with security levels. In order to protect the information confidentiality, the following properties must be satisfied: no-read-up and no-write-down. No-read-up means that a subject can only access and read from an object if the access class of the subject dominates the access class of the object. No-write-down means that a subject can access and write to an object if the access class of the subject is dominated by the access class of the object.

If these properties are satisfied, then the information flow from high level subjects/objects cannot flow to subjects/objects of lower or incomparable levels. It is important to ensure both properties are satisfied. The no-read-up is obvious, i.e. subjects not cleared to access some object cannot read it and no information is leaked. The second property, no-write-down, is also important to prevent a subject with clearance to read an object from modifying a second object with a lower security level, which can later be read by a subject with clearance to read that second object but not the first. Although these properties prevent a dangerous flow of information, they can be too restrictive and in a real scenario objects might have to be downgraded.

The mandatory policies based on integrity are based on a model presented in (Biba, 1977). In this case the goal is to prevent subjects from modifying information in objects they should not write to. Like the mandatory policies based on secrecy, each subject and object are associated to an access class with an integrity level and a set of categories. The integrity level in this case refers to the trust put on the subject to modify and insert sensitive information. To guarantee that the integrity of the information is maintained, the following properties must be satisfied: no-read-down and no-write-up. No-read-down means that a subject can only access an object to read if the access class of the subject is dominated by the access class of the object. No-write-up means that a subject can only access an object to write if the access class of the subject dominates the access class of the object.

This model prevents information stored in low objects (less reliable) to flow to higher, or incomparable, objects. These properties are the dual of the two properties in the secrecy based mandatory policies.

2.1.3 Role-Based Access Control

In this section we will present RBAC. RBAC (Ferraiolo, 1992) is a type of access control that restricts the ability of a group of subjects to access or perform some operation on a system. The idea is that in many organizations, users normally don't own the information they have access to but the organization does. A user in such an organization usually has access to some information because of their job, also referred to as their role.

So, members of that organization are assigned particular roles, e.g. Teacher or Security, and through those role assignments they acquire the computer permissions to perform the operations they need. Since users are not assigned to the permissions directly, but rather acquire them through their role (or roles), the cost of management of individual user permissions becomes a matter of assigning appropriate roles to the user's account. This simplifies common operations, such as adding a user, or changing a user's department. For each user, the active role is the role that he is using, but he can be a member of several roles (see Figure 1).

There are three primary rules that are defined for RBAC: 1) role assignment, i.e. a user can only exercise a permission only if and only if the user has selected or been assigned a role. 2) Role authorization, i.e. a user's active role must be one that was authorized for him. With the first rule, this rule ensures that users can only use roles that they have been authorized to use. And 3) permission authorization, i.e. a user can only exercise a permission if it is authorized for the user's active role. With the first two rules, this rule ensures that users can only use permissions for which they are authorized.

A role can be thought as a set of transactions that can be performed on some resources, and any user assigned to that role is capable of performing those transactions given that it is the active role for the user's session.

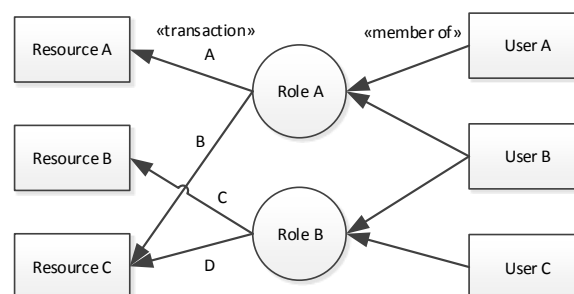


Figure 1. Role relationships.

The transactions can be more than simple read and write access to resources, e.g. a teller in a bank might be able to perform a transaction to make a deposit into some client account, which requires permissions to read and write certain fields in the account file and the transaction log, but another

transaction might be available to the accounting supervisor that corrects transactions, which requires the same read and write permissions, but the process executed and the values written into the files are different.

Sandu et al.(Sandhu, 1996) introduced the model known as RBAC96, which provides a family of models from which other variants can be built upon. Figure 2 shows the most complete model of the family (RBAC₃) that contains the concepts used to build the family of models, which include: users and roles, permissions, sessions, role hierarchy and constraints.

Users are human beings and the roles are a collection of users and a collection of permissions, it just serves to connect the two collections. The roles can also create a hierarchy, where roles can have child roles. A role with child roles works as if it possesses every permission that is also possessed by its children.

Permissions are the approval of a specific access method to one or more resources in the system. They allow the users that have roles associated with them to perform operations on the system. The range of the allowed access can vary from many resources, to a single property of a single resource.

A session is a semi-permanent dialog between two or more communication devices. When a user creates a session he can select from the roles he wants to use (the active roles). The roles can be used by more than one user at a time.

The role hierarchy is something that many applications have and allows the propagation of permissions from the lower level roles to the upper level roles.

Constraints are fundamental for RBAC that allow more complex organizational policies to be created. It can be applied to the relation between users and roles, roles and permission or even in the session mappings.

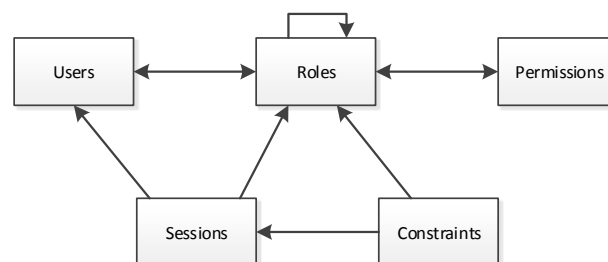


Figure 2. RBAC96's proposed RBAC₃ model.

The most often mentioned constraint is the mutually exclusive roles. If two roles are mutually exclusive, then a user can belong to none or one of them, but not both. This allows to implement separation of duties, which is important to prevent fraud and errors. Other constraints involve restricting the cardinality of users that are members of a given role, requiring that a user is a member of some role in order to be a member of another and even temporary restrictions, such as constraining the number of sessions a user can maintain.

As mentioned, Sandu et al (Sandhu et al., 1996) defined a family of models, ranging from RBAC₀ to RBAC₃. RBAC₀ is the base model, consists of every component present in Figure 2 except for the role hierarchy and constraints, RBAC₁ consists of the elements present in the RBAC₀ base model including role hierarchy, RBAC₂ consists of the elements present in the RBAC₀ base model including constraints and RBAC₃ combines both RBAC₁ and RBAC₂ models to provide a model with every component.

Barka et al. (Barka, 2000) defines the action of delegating roles in RBAC. Delegation is the ability a user can use to authorize a non-member user to become a member. This can be useful when a member of one role is required to perform some operation that is not authorized for that role, e.g. to allow a teaching assistant to grade some homework. It is important to revoke the delegation so that the delegated user doesn't keep the permissions longer than necessary.

One approach consists of defining a time limit for the delegation so that after the defined time it becomes automatically revoked. It has the advantage of being a simple way to ensure the delegation will be revoked, but it doesn't provide enough security, if not supervised the delegated user can behave badly which can cause harm to the system and if the time is not chosen carefully the delegated user might keep the new permissions for longer than necessary or shorter than needed.

Revocations to delegations can also be done manually, in the case the user with the delegated role behaves badly. This can be done only by the user that delegated the role or by any member of the delegated role. The latter has the advantage of the original member, if behaves badly, to be revoked by any member immediately which reduces the damage done, but has the disadvantage of possibly raising conflicts between members of the delegated role if someone other than the member that delegated the role revokes the delegation.

Regarding the hierarchy of roles, since parent roles inherit the permissions of its children, there are some delegations that are useless and others have more risks than others. A delegation can fall under one of three categories: upward delegation, downward delegation and cross sectional delegation.

Upward delegation is when a user is delegated a child role of the role it originally belonged to. Since the parent roles inherit the permissions of its children roles, this type of delegation is useless.

Downward delegation happens when a user is delegated a role that is a parent role in relation to its original role. This works when a partial delegation is made. A partial delegation is the delegation of one or many children of that role. If the whole role were to be delegated we would actually shrink the hierarchy. This type of delegation effectively "promotes" the user to a higher role.

Cross sectional delegation is the most useful, for example, a member of the auditing department can delegate its role to a member of the sales department to provide it with the capabilities for auditing its own department. In this type of delegation, every member of the delegated role or of any of the parent roles can delegate that role.

2.1.4 Attribute-Based Access Control

In this section we will present ABAC. ABAC (NIST, n.d.)(Hu, 2014) is a logical access control model that is relevant because it controls access to resources by evaluating rules against the relevant environment to the request, the attributes of both entities (subject and object) and the actions. Attributes can be any property that can be defined and that have some value that can be assigned. The simplest form of ABAC checks the attributes of the subject and the object, the environment conditions and the access control rules that define the allowed operations for the combinations of these factors. Figure 3 shows a basic ABAC model where: 1) the subject requests access to an object; 2) the access control mechanism evaluates: rules (a), the subject attributes (b), the object attributes (c) and the environment conditions (d) to produce a decision; 3) if authorized, the subject is given access to the object. Every implementation of ABAC is capable of this set of functionalities. It can also enforce policies based on DAC and MAC models.

The rules or policies that can be enforced through an ABAC model are only limited by the computational language used, which means that ABAC has great flexibility. For example, a subject can receive a set of subject attributes when it is hired, e.g. Alice is a teacher in the Biology department. An object can receive attributes upon creation, e.g. a door in the biology department with an electronic lock. Then the administrator creates an access control rule that controls the set of operations that are allowed in the system, e.g. all teachers can open doors with electronic locks in their department. This is flexible because if Alice ever changes departments and the same rule is applied to each one, then she will be able to open the doors with electronic locks in the new department instead of the ones in the biology department just by changing her attribute that specifies what department she belongs to, no modifications to existing rules or object attributes are required. This benefit is often referred to as accommodating the external user and is one of the primary benefits of employing ABAC.

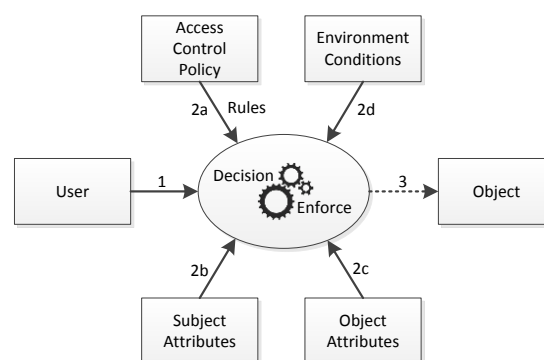


Figure 3. Basic ABAC model.

In short, ABAC describes attributes for subjects and objects that are managed by an access control rule set that dictates what operations are permitted. This allows administrators to apply access control policies without knowing the specific subjects and for any number of subjects that might require access. When new subjects integrate the organization the rules and objects do not need to be

changed provided that they get the proper set of attributes to allow them to access the required objects.

2.2 Relational database management systems

In this section we will present an overview on RDBMS, the techniques that are used to enforce security policies in them and the current tools used to connect client applications with them.

This section is divided as follows: Section 2.2.1 will present the relational model, section 2.2.2 will present the different mechanisms that exist for policy control enforcement and section 2.2.3 will present current tools to access data stored in RDBMS from applications.

2.2.1 Relational model

In this section we will present the relational model used in RDBMS. A DBMS is a set of programs that manages and provides ways to access the stored data (Sumathi, 2007). The main objective of a DBMS is to provide: data availability, data integrity, data security and data independence. Data availability means that the data must be available to many users in a convenient format and at a low cost. Data integrity means that the data cannot be corrupted and become unreliable. Data security means that only authorized users can access the data and conflicting changes made to the same data must be resolved. Finally, data independence means that the manner of how the data is stored shouldn't be an issue for the users, whom must be able to see the data in an abstract way.

Additionally, a DBMS can provide mechanisms to perform data recovery and to provide concurrency access to the data. Data recovery is needed when the storage medium, like a hard disk drive, fails or the DBMS process/operating system crashes while an operation is being performed. In these cases the DBMS can use many techniques, like using a log file to record the operations done and a checkpoint to indicate the operations that were committed to disk. This allows a DBMS to have an indicator of what was committed to disk and which operations must be redone, i.e. the operations successfully completed after the checkpoint and before the failure. The concurrency mechanism can be supported through several techniques, of which we emphasize: locks and timestamps.

Locks are used to "lock" an item, like a table or a table row, preventing other operations from accessing that item while it is being written. Locks can be binary (exclusive) or shared. With binary locks the item is either locked or unlocked, with the shared locks many operations can the item (sharing its lock) but only one can possess a lock when it intends to write on it.

Timestamps can be the system time or a logical counter and it allows operations to check the "age" of the item they are accessing. Items have a timestamp for the latest read and the latest write operation performed on them. When reading, if the timestamp of the operation indicates that it is older than the latest write operation on the item then the operation must be rejected, otherwise it is allowed. When writing, if the timestamp of the operation indicates that it is older than either the read or write operation's timestamp on the item then the operation is rejected, otherwise it is allowed.

S-DRACA concerns a very specific type of DBMS: the relational database management system (RDBMS). The relational model was introduced by E. Codd (Codd, 1970) and a relation is a set of tuples in a table, where they are organized in rows. Each tuple can have several elements that are a member of a data domain, also known as the data type. The data domains, together with an attribute name, make the attribute value for an element. These are represented by the columns in the table, as demonstrated in Table 3.

Table 3. Example relation in a RDBMS.

ID	Product	Stock	Price
1	Bread	10	0.50
2	Yogurt	15	0.89

This table stores products information and we can see that there are four attributes in the relation (*ID*, *Product*, *Stock* and *Price*). Each of these attributes has a type associated with them, i.e. the *ID* attribute can be represented as an integer and the *Product* as a string. The *ID* attribute is special in this relation because it functions as a key that uniquely identifies a tuple (i.e. a row) in the table. There are two types of keys: primary keys and foreign keys. Primary keys uniquely identify a tuple in a table and can consist of any number of attributes. The *ID* attribute in the example is the primary key for that table. Foreign keys are references in a table to the primary key of other tables, e.g. a table with sale's information could have a foreign key to our product's table to associate products with sales.

The fact that the primary keys must uniquely identify a tuple in a table means that they must have a different value in every tuple. So, using the *Stock* or the *Price* attributes as the primary key isn't a good idea because they can have repeated values. The name of the product can have the same problem but it has another disadvantage to it. Textual attributes should not be used as primary keys because they reduce performance, the reason being that integers are compared faster than strings, which requires character by character comparison. This means that it can be difficult to find what is called a natural primary key, i.e. something about the entity being stored that uniquely identifies it and the approach of using an artificial primary key, such as an incrementing integer for an ID, is used instead.

The data in the database is kept consistent by the RDBMS by applying constraints (Codd, 1970), of which are included: domain integrity, entity integrity, referential integrity and user defined integrity. The domain integrity constraints guarantee that the values of each attribute are valid for the defined data-type and if they can be null or not. The entity integrity constraints guarantee that a primary key does not have duplicate values. The referential integrity constraints guarantee that the value in a foreign key actually exists as the primary key of the referenced table. Finally, the user defined integrity constraints are dependent of the specific domain of the data stored in the database, e.g. the price and the stock of a product cannot be negative.

After a database is created, the data is usually accessed using the Structured Query Language (SQL)(Chamberlin, 1974). SQL can also create and manipulate tables and other data such as security (users, permissions, etc.).

2.2.2 Policy control enforcement mechanisms

In this section we will present the current policy control enforcement mechanisms for RDBMS. The current mechanisms for implementing access control policies for RDBMS consists of creating a separate layer of security by following one of several possible approaches: traditional, PDP – PEP, Secondary and Approximate Authorization Model (SAAM) and some other approaches such as using views and query rewriting techniques

The traditional approach uses a security layer made available by the RDBMS through some tools that can define DAC, MAC and RBAC policies. With DAC, a creator of a relation in a database is its owner and has the ability to grant other users with the ability to access it using commands such as GRANT or to take those permissions back using commands such as REVOKE. With MAC the security levels and clearance are applied to databases. Finally, with RBAC, permissions are associated with roles and users play one or more roles in each database. These access control types are managed by the RDBMS and only if a user attempts to do some operation that is not authorized will it notice the access control policies being applied.

In the PDP-PEP (Policy Decision Point – Policy Enforcement Point, see Figure 4), there are two major components responsible for enforcing the access control: the PDP and the PEP. The PDP is an independent component responsible for deciding if an access attempt is authorized or not and is configured by security experts. To do that it evaluates the access requests against the access control policies. The PEP is responsible for intercepting requests made by users to a protected resource and enforces a PDP decision for that access authorization. This approach allows the separation of the place where the policies are defined and the place where they are enforced.

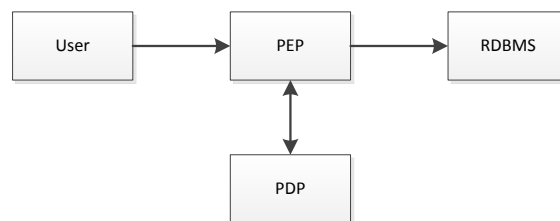


Figure 4. PDP-PEP security layer model.

Komlenovic et al. (Komlenovic, 2011) propose a distributed enforcement of the RBAC policies which aims to solve performance problems at the decision points. To reduce the system delay and improve availability, SAAM can be used. It adds secondary decision points (SDP) with cached RBAC policy rules, removing the need to query the single PDP every time an access request needs to be authorized. These access requests are made by the PEP, which is also distributed, and enforces the

decisions made by the SDPs or the PDP. One question to ask is which structure and associated algorithms should be used in the SDP in order to avoid querying the PDP. There are six candidates that were explored: directed graphs, access matrix, CPOL (Borders, 2005), authorization recycling (Wei, 2011), bloom and cascade bloom filter (Bloom, 1970).

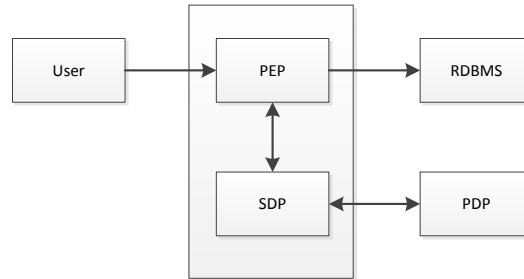


Figure 5. SAAM policy enforcement architecture.

Other mechanisms involve the usage of views (Rizvi, 2004) and parameterized views (Roichman, 2007). Views are a result set of a stored query on the data, which the database users can query just as they would in a persistent database collection object. By applying constraints on the query that creates the view, it is possible to control the data that is available to the users. Since views are standard SQL, they have the advantage of not requiring additional tools or techniques to enforce the access control policies. However, this approach is not scalable, since its number grows with the number of policies and users that have different authorizations require different views. Parameterized views accept runtime values in the select statement that defines them, helping with the lack of scalability that the non-parameterized views suffer.

Query rewriting (Rizvi et al., 2004) is also another technique used to control the users access attempts, where CRUD expressions are modified before they are executed in order to prevent access to unauthorized data. The queries are usually modified in a central server and various different approaches can be used (Rizvi et al., 2004)(Chaudhuri, 2007): addition of predicates, replace tables with views, masking cells and column removal.

The addition of predicates is an approach where predicates are added to the CRUD expressions in order to filter out the data that should not be disclosed to the users. The replacement of tables with views is self-explanatory. The tables accessed in the select statements are replaced by views, which provide only authorized data. The masking of cells is a technique where CRUD expressions are modified in order to change the protected values with either named variables, the best solution but not supported by all RDBMS, or the SQL null values, which has the problem of becoming mixed with real null values. The removal of columns technique removes protected columns from the returned records by erasing them from the CRUD expressions. This, however, means that users with different authorizations will have access to relations with different schemas and errors occur if the user tries to access a removed column.

The query rewriting technique has the advantages of being transparent to the user, since they use CRUD expressions as if no security policies were enforced, and is scalable, since the number of

needed CRUD expressions per application remains the same unlike the usage of views. However, because the users are unaware of the policies enforced, when a query is discarded for violating a security policy, the users themselves have to correct it, sometimes without knowing exactly what the problem was. There is also the performance decay that comes with the fact that a central server is used to rewrite the CRUD expressions.

2.2.3 Application's database access libraries

In this section we will present various tools and libraries used by programmers to access data stored in DBMS from applications.

This section is divided as follows: Section 2.2.3.1 will present the Java Database Connectivity, section 2.2.3.2 will present ADO.NET, section 2.2.3.3 will present Hibernate and finally section 2.2.3.4 will present the Language Integrated Query.

2.2.3.1 Java Database Connectivity

In this section we present Java database connectivity (JDBC)(Oracle, 1997a), which is a Java-based technology that allows applications to access data in a database. This technology is an API for Java and defines how a client may access the database. It provides methods to for querying, inserting, updating and deleting data in a database. JDBC is oriented towards relational databases.

Figure 6 presents the JDBC architecture. JDBC provides applications with an API which allows them to access the data stored in a multitude of different DBMS. To handle the different communication protocols used by the different DBMS, JDBC uses a Driver Manager that manages the drivers used to access them. Normally each DBMS provides its own JDBC driver.

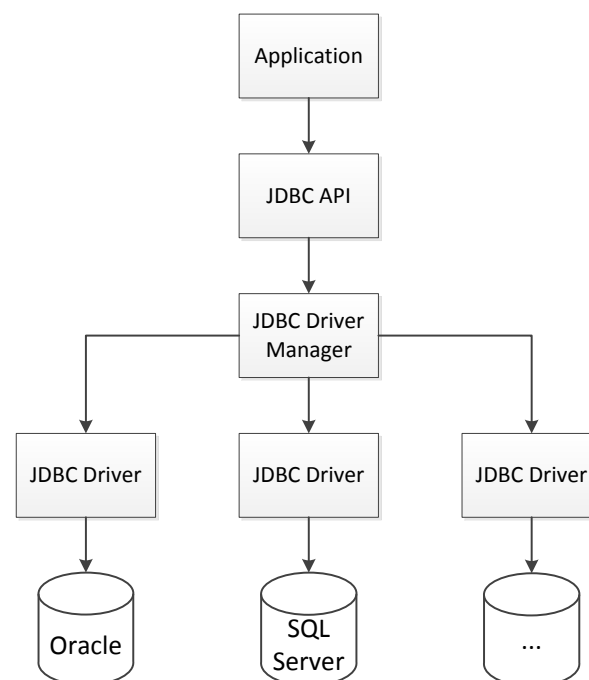


Figure 6. JDBC architecture.

JDBC can have many multiple driver implementations and they can be used by the same application, each implementation is responsible to connect to a specific DBMS (e.g. MySQL, SQL Server, etc.). The API also provides a mechanism for loading dynamically the correct Java packages and registering them with the JDBC Driver Manager. The Driver Manager is then used by the applications as a factory to instantiate JDBC connections.

A Connection object is responsible for creating and executing data manipulation language statements, which can be something like a select, an update, an insert or a delete statement, or data definition language statements, such as a create statement. Additionally, stored procedures can also be invoked.

A statement can be represented using one of three classes: a Statement, a PreparedStatement or a CallableStatement. The Statement class sends the SQL statement to the database each time it is executed (see Figure 7). With a PreparedStatement, the statement is cached and the DBMS builds an execution plan that becomes pre-determined, allowing it to be executed many times in a more efficient way (see Figure 8). The CallableStatement is used to execute stored procedures defined in the database (see Figure 9).

Insert, update and delete statements return the number of rows affected by the operation. Select statements return a result set, which are local data sets (LDS) with each row returned by the database. They are used to walk over the rows, whose values in each column can be retrieved (see Figure 10). The result set knows the names of the columns and their data types from the metadata information it receives.

```

42      try (Statement stmt = conn.createStatement()) {
43          ResultSet rs = stmt.executeQuery(
44              "SELECT * FROM Orders WHERE CustomerID = 'GREAL'");

```

Figure 7. JDBC statement usage example.

```

64      try (PreparedStatement pstmt = conn.prepareStatement(
65          "SELECT * FROM Orders WHERE CustomerID = ?")) {
66          pstmt.setString(1, "GREAL");

```

Figure 8. JDBC prepared statement usage example.

```

119     try (CallableStatement cstmt = conn.prepareCall(
120         "[dbo].[Ten Most Expensive Products]")) {
121         ResultSet rs = cstmt.executeQuery();

```

Figure 9. JDBC callable statement usage example.

```

49     while (rs.next()) {
50         System.out.println("OrderID: " + rs.getInt(1));
51     }

```

Figure 10. Result set row iteration.

The JDBC API has two paramount sets of interfaces: one for the application side, which was described previously and another for the lower-level driver. The JDBC drivers are components that

enable Java applications to interact with a database. Most databases have a JDBC driver available, allowing Java applications to access a wide variety of RDBMS solutions.

2.2.3.2 ADO.NET

In this section we present ADO.NET (Microsoft, n.d.), which is a set of classes that expose services to .NET Framework developers. It provides a consistent access to data sources such as databases and others such as XML files or exposed through an ODBC interface. It allows to execute query statements like JDBC: it connects to a data source using a connection object (e.g. a `SqlConnection`) and can execute query statements or create the equivalent to prepared statements in JDBC, named '`SqlCommand`'. Executing a select statement returns a reader that can be used to iterate over the rows of the relation returned by the data source. Like JDBC, it is more focused on compatibility rather than the access control policies defined in the data source, so the developers must master the database schema and the defined access control policies when using ADO.NET manually.

2.2.3.3 Hibernate

In this section we present Hibernate (JBoss, 2001), which is an object-relational mapping library for Java that provides a mapping between the object-oriented domain model and the relational paradigm in databases. Java classes can be created from the database relational model or vice-versa by some IDEs and tools that automate the process, but the library itself requires the creation of the database tables, java classes and configuration files/annotations. It provides a persistence engine, which transparently does persistence of java objects that have been mapped into a database table. It can be seen that the automated mapping only takes into account the access control policies at the time the mapping was performed and any changes to them requires the modification of the java objects and their mapping configuration. Other related frameworks/libraries include the Java Persistence API (Oracle, 2006) and EclipseLink (Eclipse, 2008).

2.2.3.4 LINQ

In this section we present the Language Integrated Query (LINQ), which is a Microsoft .NET framework that extends .NET languages to add query expressions, similar to SQL statements, which can be used to extract data not only from databases, but arrays, enumerate classes and other data sources as well (Microsoft, 2007).

It is an object-relational mapping which models a relational database using .NET classes, and then LINQ can be used to query the database using the classes, displaying the tables and columns that are available. Furthermore, it fully supports transactions, views and stored procedures.

Next we demonstrate how LINQ can be used:

```
var orders = from o in db.Orders
              where o.Order.ShipCountry == "USA"
              select o;
```

This LINQ query selects all orders whose ship country is USA. Note that because .NET classes are created to map the relational database, the developer does not need to master the database schema. However, if the access control policies are altered the application will not be aware of that fact until errors occur in runtime.

2.3 Related work

In this section we present the work related to access control enforcement and their major contributions in order to understand the current state of art.

Ur/Web

Chlipala et al. (Chlipala, 2010) present a new tool, Ur/Web, which allows programmers to write CRUD expressions that can check statically the access control policies in a system backed by a DBMS. In this work, each data that can be accessed is determined by each policy. Then, programs are developed and checked to make sure that the data involved in the CRUD expressions is accessible through some policy. For the policies to be able to vary from user to user, the queries that check them can use actual data and a new extension to the standard SQL to capture ‘which secrets the user knows’. This extension is based on a predicate referred to as ‘known’ which models the information users are already aware of to decide what information is to be disclosed.

Consider the table user with each user’s name and password. A policy can be defined using the following syntax:

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

In this case, the policy *sendClient* prevents users from reading data that belongs to other users by allowing them to read the rows to which they know the password.

It is a solution with potential, but it doesn’t check the access control on *where* clauses allowing queries to leak protected data implicitly and since the process of validation takes place during compile-time, the programmers still have to master the database schemas and the defined security policies when the source code is being written.

Integrating access control policies within database development

Abramov et al. (Abramov, 2012) present a complete framework that allows security aspects to be defined early in the software development process and not at the end. They present a model from which access control policies can be inferred and applied.

Their contribution with their work is a new methodology, from which the security patterns are clearly defined using common modeling techniques. Since the security patterns are clearly defined, it is possible to enforce them over application designs and to generate the required artifacts from the transformation rules that define how to go from the application model to the database code.

Nevertheless, similarly to (Chlipala, 2010), the validation process takes place only at compile time, again requiring the programmers to master the established access control policies.

Hippocratic databases

Hippocratic databases are databases that are designed to integrate privacy policies into their architecture. Ten principles (Agrawal, 2002) have been announced that define Hippocratic databases: purpose specification (the purpose for which data has been collected must be associated), consent (the purpose must have the donor consent), limited collection (the data collected must be the minimal amount that satisfies the purpose), limited use (only queries that are consistent with the purpose of the collected data may be run), limited disclosure (the information shall not be communicated outside the database for other purposes than those consented), limited retention (the data collected shall only be retained as long as necessary to satisfy the purpose of the collection), accuracy (data must be accurate and up-to-date), safety (data must be protected against theft and other unauthorized access by security mechanisms), openness (the donor of some data must always be able to access all of it) and compliance (the donor must be able to verify the compliance of the principles). These principles have been attempted to be used in practice like in (Padma, 2009) for PostgreSQL. The next example is based on Hippocratic PostgreSQL:

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

This Select query produces a result that has its columns restricted for the combination of the purpose and recipient. Additionally, it will be restricted to only contain data to be shared with the purpose of auditing. We can see from this example that databases with Hippocratic principles address a different kind of access control, i.e. privacy, which lets the owners of some information specify how, when and for what their information can be accessed for.

(LeFevre, 2004) presents an approach to limit the disclosure of data in Hippocratic databases. It employs the query rewriting technique and the policies are defined with either EPAL (Ashley, 2003) or P3P (Cranor, 2002) and it describes rules that are used to specify whom may access the data and for what purpose. In short, the application first submits the query to the database and retrieves the result. Then, the application goes through the returned records and filters out prohibited information. The policies can either be enforced at: the table level, where each table is associated with a view that

replaces prohibited cell with null values or at the CRUD expression level, where the records returned from a select statement are filtered to remove the prohibited values.

SESAME

SESAME (Zhang, 2003) is a dynamic context-aware access control mechanism for pervasive GRID applications. It complements current authorization mechanisms by dynamically granting and adapting permissions to users based on their current context. The dynamic role based access control model, which extends the classic RBAC model, is the base model used in SESAME. When subjects log in, it assigns a default role hierarchy and then monitors the context of the subjects and delegates roles as needed. There are two types of contexts: the object context and the subject context. The object context has information such as user's location, time, local resource and link state. The subject context contains information such as the system's current load, connectivity to a resource and availability. However, SESAME enforces the access policies in a centralized system which has the drawbacks in terms of stability and performance.

SELINKS

SELINKS (Corcoran, 2009) extends the LINKS (Cooper, 2007) programming language similar to LINQ that can be used to construct secure web applications. A developer using LINKS could write a program and the compiler would then create the byte-code for each tier of the application and for the security policies as well. These are then encoded on RDBMS as user-defined functions, which check during runtime what actions each user is allowed to perform. The access to the data can be mediated using functions that enforce the policies by accessing security labels, i.e. types that define the metadata, defined by the programmers. A type system named Fable is used to ensure that the data is accessed only after consulting the proper policy enforcement function.

SELINKS has the advantage of integrating a security context in all the application tiers, using Fable to prevent the avoidance of the security policies, which is optimized to reduce network load by using user-defined functions in the database to check the permissions instead of transferring the required data to perform that check in the web servers. Furthermore, it's a single tool, easing the development process by requiring the programmers to learn just one tool. However, the security labels define a group-based access control policy that only distinguishes between read and write operations, hence it is not possible to apply restrictions on the CRUD expression level.

Jif

Jif (Zhu, 2012) is a security-typed programming language that extends Java to give support for information flow control and access control, enforced at both compile time and runtime. Java is extended to support the addition of labels that can express the access control policies to be enforced and how the information may be used. The following variable declaration declares not only that the variable *x* is an integer, but also that the information in *x* is managed by a security policy:

```
int {Alice→Bob} x;
```

In this case, the label expresses that the information in x is controlled by the principal Alice and that Alice allows this information to be seen by the principal Bob. The other direction in the label, $\{Alice←Bob\}$, means that information is owned by Alice, and that Alice permits it to be affected by Bob. With these labels, the Jif compiler is able to analyze the information flow within programs to determine whether they enforce policies expressed by the labels or not. Jif supports labels and principals, as shown, but also principal hierarchies, integrity and confidentiality constraints, authority delegation between principals, confidentiality (declassification), integrity (endorsement) downgrade and a form of label polymorphism. However, the language is mostly used to manage the information flow at the application level and not to access data in RDBMS, so other tools must be used to compensate.

Reflective Database Access Control

In (L. E. Olson, 2008), the Reflective Database Access Control was presented. It is a model in which CRUD expressions are the privileges in the database rather than using static privileges defined in the ACLs. To express the reflective access control policies the Transaction Datalog (Bonner, 1997) was used. An implementation of this access control model was presented in (L. Olson, 2009).

Security-driven model-based dynamic adaptation

A security-driven model-based dynamic adaptation is presented by Morin et al. (Morin, 2010) to address a problem where even with the separation of the policies and the application code that is done in theory, it is never fully done in practice, leading to some rules being written directly in the application code. The approach uses meta-models that describe the access control policies and the application architecture and defines how to map (statically and dynamically) the access control policies meta-model to the application architecture meta-model. It does not address how to statically implement secure and dynamic security mechanisms.

Java EE

Java Enterprise Edition (Java EE) is an extension of Java (the standard edition) to build enterprise software (Oracle, n.d.-c). It uses the `@RolesAllowed` annotation to enforce RBAC policies directly at the methods level, controlling who has the permission to invoke them. It does not, however, identify who is invoking the protected methods, meaning that any user on the allowed roles can get access to the protected method. Furthermore, the process of checking if a user calls a protected method to which he is not authorized to is only done at runtime, which means that developers have no way to statically validate if their code respects the access policies.

Annotated objects

In (Fischer, 2009), Object-sensitive RBAC (ORBAC) is an extension of RBAC that can be used with object-oriented programming languages. It attempts to address the shortcomings in the current RBAC

model and other frameworks like the ones just discussed. It controls the access at the level of single objects allowing a fine grained control. Unlike some frameworks, Object-sensitive RBAC allows developers to write access code knowing if they are violating any access control policy or not through its type system.

Zarnett et al. (Zarnett, 2010) present a different solution, which can be applied to control the access to methods of remote objects via Java RMI (Oracle, n.d.-d), a framework that allows an application to use objects that exist in a different application, possibly in a different machine. The server, which hosts the remote objects, enriches the methods and objects with metadata about the roles that are authorized to use them through the usage of Java Annotations. Then, RMI Proxy Objects handle the requests to execute methods on the remote objects hosted in that application, which are generated in accordance with the established access control policies (they contain the authorized methods only) for the role of the user requesting the access.

Fischer et al. (Fischer et al., 2009) present a more fine-grained access control, which uses parameterized Annotations to assign roles to methods, and the work presented in (Zarnett et al., 2010) defined each annotation required for the domain of the application. These approaches, in contrast with our concept, do not ease the access to a relational database since the developers still need to acquire a deep understanding of the database schema and also the defined access policies to access database objects.

Predicated grants

In (Chaudhuri et al., 2007) is proposed a fine-grained authorization based on adding predicates to grants and an extension to the current SQL authorization model in order to support it. It basically allows to define which records in a table a user is able to access, the value returned to the public, etc. As an example we can have:

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

This authorization specifies that each employee can access its own employee information and any other information is nullified so that it cannot be accessed.

This model addresses aspects such as cell-level security by nullifying values, allowing predicates to be added to any kind of grant (CRUD expressions, stored procedures and functions), authorizing aggregation functions while preventing access to the base data and even mechanisms to manage large number of users.

λ DB

λ_{DB} is a programming language, presented in (Caires, 2011), that enforces access control policies to data by static typing for data-centric programs. It allows data structures to be defined, known as *entities*, which are checked at compile-time against the access control policies and other information

that is dependent on the context. A permission in this approach is comprised of: the action granted (read or write), the attributes of the entity and a logic condition. An example of such a permission is:

```
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 has four attributes (userid, public, photo and secret) and the conditions shown state that the public attribute can be always read, the secret can only be read by its owner and the photo attribute can be read by its owner and its friends. The write condition applies to all attributes and only allows the owner to write the attributes.

This approach provides only a single action to authorize update, insert and delete operations on the attributes, not allowing to distinguish between them, but in contrast to other solutions, like Ur/Web, the *where* clauses can be protected.

Assurance management framework

A similar approach to our work was presented by Ahn et al. (Ahn, 2007), where a tool is used to define a security model from which is generated some source code that checks if there is any violation of any access control policy. This verification process only takes place the source code has been written, this way not addressing the key aspects of our work.

Extensible access control markup language

The extensible access control markup language (OASIS, 2010) is a declarative access control policy language implemented in XML and a processing model that describes how access requests are evaluated according to the rules defined in policies. It uses the PEP-PDP model to enforce the access control policies, so the PEP component communicates with the PDP and, when the authorization is granted, it uses some static business logic to complete the task. When a modification is made to the underlying policies the logic in the PEP does not adjust automatically, so they must be modified in advance.

Other works

There are other works related to access control enforcement: Jayaraman et al. (Jayaraman, 2013) propose a new technique and a tool to detect errors in the RBAC policies and, finally, Wallach et al. in (Wallach, 2000) propose new semantics for stack inspection that solves issues with the traditional stack inspection, which is used to detect a dangerous call (e.g. to the file system) and verify if it is allowed. Our work complements these, in regards to the access to relational databases, by generating static access control mechanisms automatically that are in accordance the established RBAC policies,

this way freeing the programmers from mastering the database schema or the defined access control policies.

The works presented in (ÓM Pereira, 2012) and (ÓM Pereira, 2013) deal with the direct and the indirect access modes, but none defines exactly how the RBAC policies are to be enforced from the CRUD expressions. The work presented in (ÓM Pereira et al., 2013) can be seen as the initial approach to achieve the goals of the work presented in this work. It focuses on the CRUD expressions and on both access modes but does not convey how to connect the CRUD expressions and the RBAC-based policies. The work presented in (ÓM Pereira et al., 2013) also leverages (ÓM Pereira et al., 2012) but it is mainly focused on addressing a distinct key aspect regarding security where the runtime values used on the direct and on the indirect access modes are also regulated by the access control policies.

2.4 Summary

In this section we presented the state of the art regarding access control policies, the DBMS, the existing methods to implement access control mechanisms and the tools used to access them. Then we presented some related work and how can S-DRACA complement them..

3 Technological Background

In this section we will discuss the technologies that are still used or that were introduced to develop S-DRACA, namely Java and some of its functionalities and various aspects of cryptography and authentication. We will also look into the service coordination aspects and languages to be able to implement the pre-defined sequences of CRUD expressions that the client must follow.

This section is divided as follows: Section 3.1 will present some notions of service coordination, section 3.2 will present several Java functionalities that were essential for the development of S-DRACA, section 3.3 will present some information about computer security in the areas of cryptography and authentication and section 3.4 will summarize the contents of this section.

3.1 Service coordination

In this section we present some notions of service coordination. Its purpose is to use them as a base for the implementation of the sequences to be enforced in S-DRACA, which aims to restrict the operations a user could perform to the database. The restrictions are enforced, even when the user has the permission to perform those operations, in a way that he has to follow a predefined sequence of operations. This can be seen as coordination of services and so we will analyze the current technologies that provide this type of functionality.

To encode the sequences in a way that all information required to provide the proposed functionalities is available, we could have used two of the technologies that are used to control services workflow: 1) service orchestration, which requires every service to be requested by a central control point, and 2) service choreography, which allows a service to request the next service.

Standard languages for each technique were proposed. This section is divided as follows: Section 3.1.1 presents the orchestration method for coordinating services and section 3.1.2 presents the choreography method.

3.1.1 Orchestration

In this section we present orchestration, a service coordination method that can be described as the automated arrangement, coordination, and management of complex computer systems, middleware, and services. Figure 11 shows a basic implementation of the service orchestration scheme. An application makes a request to a central control point, i.e. the orchestrator, which executes the requests, calling the different services as needed.

A solution based on this scheme would require a central component that would provide the application with the next authorized CRUD expression in the sequence.

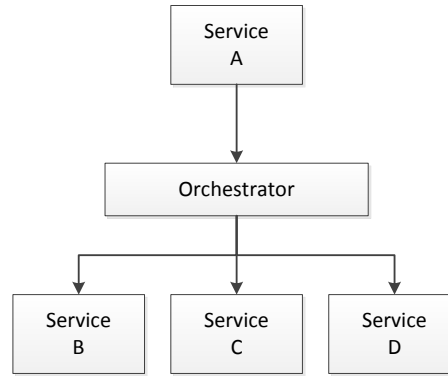


Figure 11. Service orchestration.

OASIS defined a standard language, which is still very active, called Web Services Business Process Execution Language (OASIS, n.d.). It provides a set of functionalities that largely exceeds our needs, therefore we will use some functionality similar to those provided by it but tailored to our specific needs, such as graphs and life-cycle operations of active entities.

Several other languages exist, such as Yet Another Workflow Language (“YAWL,” n.d.) and XML Process Definition Language (“XML Process Definition Language,” n.d.), but they clearly would not bring any advantage to our case.

3.1.2 Choreography

In this section we present choreography, a service coordinator method that can be described as the interaction protocol used by the services from a global perspective, i.e. every service calls the next service as needed, without the intervention of a central control point. The next service that needs to be called is defined in the protocol that is defined globally. A basic implementation of this scheme is shown in Figure 12. A solution based on this scheme would require a way to retrieve a reference to the next authorized CRUD expression in the sequence from a previous one.

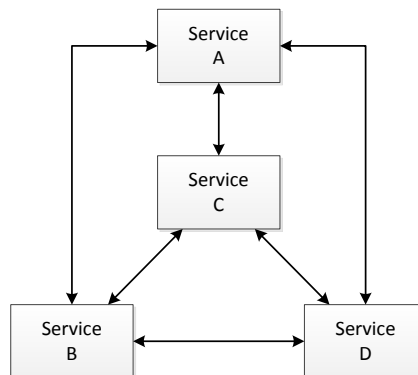


Figure 12. Service choreography.

The Web Service Choreography Description Language (W3C, n.d.) is a language from W3C aimed at describing choreographies using the global view of the observable behavior of web services.

However, the W3C Web Services Choreography working group was closed in 2009, leaving the language just as a candidate recommendation.

3.2 Java

In this section we present several functionalities of Java that were critical for the development of S-DRACA. Java is a computer programming language that is concurrent, class-based, object-oriented, and has as few implementation dependencies as possible. It is intended to enable the code that runs on one platform to run on another without having to be recompiled. Java applications are typically compiled to bytecode (class file) that can be interpreted on any Java virtual machine (JVM) regardless of computer architecture. Java was originally developed by James Gosling at Sun Microsystems (which has since merged into Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++, but provides less low-level functionalities when compared to any of them.

There were five primary goals in the creation of the Java language(Oracle, n.d.-a): 1) it should be "simple, object-oriented and familiar", 2) it should be "robust and secure", 3) it should be "architecture-neutral and portable", 4) it should execute with "high performance", and finally 5) it should be "interpreted, threaded, and dynamic".

It's these goals that led to the choice of Java as the programming language to implement S-DRACA, specially the third goal that states the solution should be architecture-neutral and portable, which made Java a programming language that runs on its own virtual machine which is available for many operating systems. We will now analyze the language's libraries and frameworks that were used in S-DRACA.

This section is divided as follows: Section 3.2.1 presents the reflection functionality and section 3.2.2 presents the functionality to write custom annotations.

3.2.1 Reflection

In this section we present Reflection, which is the ability that allows a computer program to examine and modify the structure and behavior of the program at runtime, like values, metadata, properties and functions(Malenfant, 1996). It should not be confused with type introspection, which provides only the ability to examine said structure and behavior. Reflection is commonly used in high-level languages like Java(Oracle, n.d.-e) and C#("Reflection (C# and Visual Basic)," n.d.).

Using its reflection API, Java allows the inspection of classes, interfaces, fields and methods at runtime, even without knowing their names at compile time. Furthermore, it is also possible to instantiate new objects and even to invoke methods. Figure 13 shows how Reflection can be used in Java to instantiate objects and invoke methods. First, we get a Class object that represents the class we require using its fully qualified name (line 10), which allows us to retrieve any declared members, including modifiers and parameters. The example in the figure instantiates a new String object by searching the wanted constructor using the parameter types (line 11) and using it to instantiate an

object of that class (line 12). Finally we search for the split method by its name and parameter types (line 13) and invoke it, receiving the result from the split operation (line 14).

```
10 Class<?> c1 = Class.forName("java.lang.String");  
11 Constructor constr = c1.getDeclaredConstructor(c1);  
12 Object newInstance = constr.newInstance("Big String");  
13 Method declaredMethod = c1.getDeclaredMethod("split", c1);  
14 Object ret = declaredMethod.invoke(newInstance, " ");
```

Figure 13. Instantiation and invocation using reflection.

We are required to use the parameter types to search for methods and constructors because of the method overloading feature, which allows us to declare methods with the same name using different parameters types.

Reflection is a powerful technique and a fairly advanced feature that allows us to perform operations that would be impossible otherwise. Among its features and applications, we emphasize: extensibility features, class browsers, visual development environments, debuggers and test tools. Extensibility features allows an application to use external, user-defined classes to create instances of extensibility objects using their fully-qualified names. A class browser can use reflection to enumerate the members of classes. IDEs can also benefit from the information available about the classes to aid the developer to write correct code and provide other features like IntelliSense, i.e. auto-completion tips Debuggers need to be able to inspect the private members of a class so that we can successfully debug an application. Test frameworks can make use of reflection to discover a set of APIs defined in a class and insure a high level of code coverage in a test suite.

Reflection does, however, have some drawbacks and so it should not be used indiscriminately. If an operation can be performed without the use of reflection, then it is preferable to do so. Its major drawbacks are: performance overhead, security restrictions and the exposure of internals. Since reflection uses types that are dynamically resolved certain JVM optimizations cannot be performed. The result is that reflective operations have a lower performance when compared to their non-reflective counterpart. Regarding the security restrictions, when an application that uses reflection runs in a restricted security context, e.g. in an Applet, a runtime restriction that might not be present is required. Finally, the exposure of internals occurs because reflection allows applications to access private fields and methods. Furthermore, it allows to perform operations that would be illegal in non-reflective code. This may lead to unexpected side effects, render it dysfunctional and even destroy the application's portability. This fact breaks abstractions and therefore its behavior may change with platform upgrades.

3.2.2 Annotations

In this section we present the annotations functionality. In the Java programming language, an annotation is a form of syntactic metadata that can be added to Java source code(Oracle, n.d.-f).

Many of the intended use cases for annotations involve having separate files hold information that is used to generate new derived files (source files, class files, deployment descriptors, etc.) that are logically consistent with the base file and its annotations(Oracle, n.d.-g). In other words, instead of manually maintaining consistency among the entire set of files, only the base file would need to be maintained since the derived files are generated.

Java has always had ad hoc annotation mechanisms, e.g. the transient modifier could be used to indicate that a variable should be ignored during the serialization process and the “@Deprecated” javadoc tag is another one that indicates that a method should no longer be used. Annotations do not directly modify the semantics of a program, but they do affect the way programs are treated by tools and libraries. One example of a framework that uses annotations to change a program’s behavior is JPA(Oracle, 2006), as shown in Figure 14, where the declared annotations do nothing by themselves, but the JPA implementation, at runtime, receives the class object, extracts the annotations and generates an object-relational mapping(JBOSS, n.d.).

Since the Java release 5.0, the platform has a general purpose annotation facility, also known as metadata, which allows the creation of custom annotation types. The facility consists of a syntax for declaring annotation types, a syntax for annotating declarations, APIs for reading annotations and a class file representation for annotations.

Annotations are processed at compile time by an Annotation Processor that can be extended to one’s needs. It provides facilities like objects that allow to create source code (IDE’s can detected these files and add them to a project as generated source code), detect special entities such as methods (a test framework can make use of annotations by having an “@Test” annotation in methods that are designed to test a project and run them at compile time) or make any other type of computation.

```

17  @Entity
18  @Table(name = "people")
19  public class Person implements Serializable {
20      @Id
21      @GeneratedValue(strategy = GenerationType.AUTO)
22      private Integer id;
23
24      @Column(length = 32)
25      private String name;
26
27      public Integer getId() {
28          return id;
29      }
30
31      public void setId(Integer id) {
32          this.id = id;
33      }
34
35      public String getName() {
36          return name;
37      }
38
39      public void setName(String name) {
40          this.name = name;
41      }
42  }

```

Figure 14. JPA data class example.

3.3 Cryptography and authentication

In this section we will cover the main technological aspects of computer security used in this dissertation.

Computer security is a field of computer science that covers all the processes and mechanisms that protects computer-based equipment and services from unintended or unauthorized access, change or destruction. We will cover the encryption of data, the authentication and authorization of entities, i.e. users and servers, and some of the security attacks that can be performed against them.

This section is divided as follows: Section 3.3.1 will introduce the notion of hash function, section 3.3.2 will present several aspects of data encryption, section 3.3.3 will present several methods for authentication of users and section 3.3.4 will present some attacks that can be performed against a computer system.

3.3.1 Hash functions

We start the security section by introducing hash functions. Hash functions are very commonly used in computer security, hence it's important to have a good understanding of them. A hash, also known as a message digest, is a one-way function(Kaufman, 2002a). It is considered a function because it takes an input and generates a fixed-length output. It is considered one-way because it is difficult to determine the input that generated a given output. They must be deterministic, i.e. the same input must produce always the same hash value. This property is critical for many uses of this type of functions, which range from file validation, i.e. to guarantee that a downloaded file from the internet has not been corrupted, password verification by using its hash value to keep the password secret, etc.

Hash functions can also be considered cryptographically secure, but for that it must have a couple of additional properties:

- It must be computationally unfeasible to determine the message that has a given hash;
- It must be computationally unfeasible to find two different messages with the same hash.

From these properties it follows that it should be computationally unfeasible to find a different message from a given one that produces the same hash value.

Figure 15 shows an example of a hypothetical hash function being used to hash different values. We can see that the inputs *Hello* and *Hello!* Only differ from a single character, but the generated hash value is completely different. However, the inputs *Hello* and *World* generate the same hash value, which is called a collision.

Generally, it would take about $2^{m/2}$ messages to find a collision for a hash function that generates hash values with m bits. However some exploits can be found in the functions that can reduce the number of messages needed. The current hash functions still considered safe from collision detection have outputs that are greater than 128 bits, which would require about 2^{64} different messages to find a collision with no known exploits.

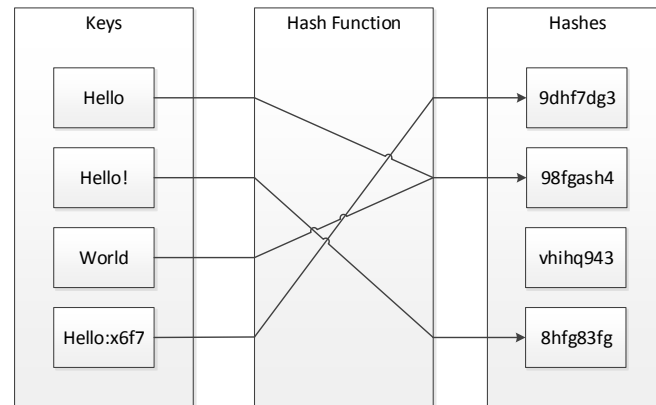


Figure 15. Example of a hypothetical hash function.

There is one use where their deterministic nature can lead to information leaking. If the passwords of users are stored in a database hashed and we see the hashed values, we can notice patterns in those hashed values. For example, when we see that two different users have the same hash value, which together with the fact that the same password has always the same hash value (deterministic property), we can deduce that both users have the same password if the probability of it being a collision is extremely low. To avoid this kinds of patterns, systems usually concatenate random data to each password, called a salt. This way if two users use the same password, different salts are concatenated and the hash values are different (or they have a high probability of being so). The salt does not need to be secret and is usually stored with the hashed password, but to detect that two users have the same password it is now required to actually be able to guess the password.

3.3.2 Encryption

In this section we will introduce several aspects regarding data encryption in computer systems. In cryptography, encryption is the process by which messages and information are encoded in such a way that only authorized parties can read it (see Figure 16). It does not completely prevent an unauthorized party from reading the data, but it reduces the likelihood of it happening. Encryption uses a sequence of bytes (key) to encrypt and decrypt the data. If the keys are short/weak then it is easier for an attacker to crack the encryption, i.e. determine the key used for the message, and read the encrypted data, whereas strong keys can take a very long time, making any attempt to do so unfeasible.

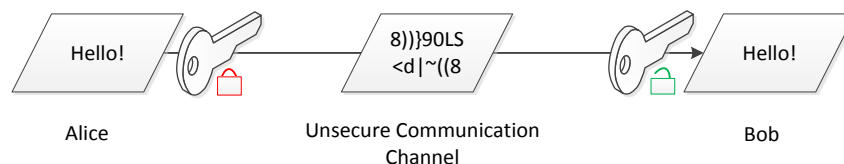


Figure 16. Basic usage of encryption to send data over an unsecure channel.

This section is divided as follows: Section 3.3.2.1 will present the usage of keys and how they can be exchanged and used for the encryption of data. Section 3.3.2.2 presents a standard protocol for data encryption.

3.3.2.1 Keys

In this section we will present the notion of keys and how they can be used for data encryption. There are two main types of keys in computer security: symmetric keys and asymmetric keys. Symmetric keys are simpler and faster to use when compared to their asymmetric alternative, since the same key is used to encrypt and decrypt the data and involves simpler mathematics. However, we have the drawback of having to exchange them between the two parties in a secure way. This drawback can be mitigated by using a key exchange protocol (e.g. Diffie-Hellman key exchange protocol), which allows to exchange a symmetric key between two parties in a secure way. Asymmetric keys do not have this drawback since it uses a key pair, one of the keys is public and can be exchanged normally, and the other key is private and is kept secret. When a message is encrypted with one of the keys then the message can only be decrypted using its counterpart. However, they are more complex to use since the algorithm used to encrypt a message with one of the keys has to decrypt the message with the other key and not with the first, involving more complex mathematics.

This section is divided as follows: Section 3.3.2.1.1 presents symmetric keys, section 3.3.2.1.2 presents a protocol for exchanging symmetric keys, i.e. the Diffie-Hellman key exchange protocol, section 3.3.2.1.3 presents the asymmetric keys and section 3.3.2.1.4 presents public key certificates.

3.3.2.1.1 Symmetric Keys

In this section we will present symmetric keys and how they can be used. Symmetric key algorithms are a class of algorithms that are used in cryptography that are able to encrypt plaintext and decrypt the encrypted plaintext using the same cryptographic key, hence the name symmetric keys. In practice, the symmetric keys represent a shared secret that two or more parties can know and that can maintain a private communication channel by encrypting and decrypting the data with it. Figure 17 represents a generic use of this type of keys. Alice and Bob possess the same symmetric key and they use it to encrypt and decrypt the messages.

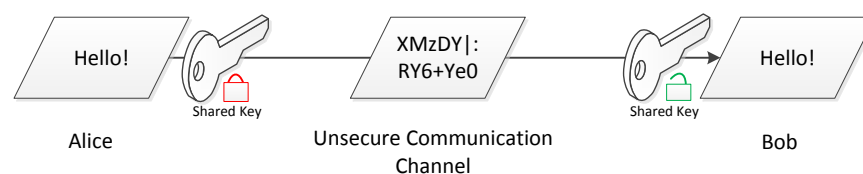


Figure 17. Message encryption over an unsecure channel using a shared symmetric key.

There are two types of symmetric-key algorithms: stream ciphers and block ciphers. Stream ciphers encrypt the messages typically byte by byte. Block ciphers take a number of bits and encrypt them as a single unit, padding the messages if necessary to match the block size.

Some algorithms that use symmetric keys include Twofish(Schneier, 1998), Serpent(Biham, 1998), AES (Rijndael)(Daemen, 1999), Blowfish(Schneier, 1998) and 3DES (i.e. DES(National Bureau Of Standards, 1999) applied three times).

3.3.2.1.2 Diffie-Hellman Key Exchange

In this section we will present the Diffie-Hellman key exchange protocol. The Diffie-Hellman key exchange is a method for exchanging cryptographic keys. It allows two parties that have no prior knowledge of each other to jointly establish a shared secret symmetric key over an insecure communications channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher.

The simplest and the original implementation of this protocol is shown in Table 4 and uses two public values: a prime number p and a generator g . The generator is a primitive root (Abramowitz, 1972) modulo p . The values a and b are Alice and Bob's secret values, respectively. A and B are the values that are calculated from the public common values (p and g) and the private values a and b , respectively, that are sent over the network. Finally, the value s is the agreed private shared secret.

Using this method both Alice and Bob arrive at the same value, because $(g^a \bmod p)^b$ and $(g^b \bmod p)^a$ are both equal $\bmod p$. Note that only a , b , and $(s = g^{ab} \bmod p = g^{ba} \bmod p)$ are kept secret. All the other values – p , g , $A = g^a \bmod p$, and $B = g^b \bmod p$ – are sent in the clear. Once Alice and Bob compute the shared secret they can use it as an encryption key, known only to them, for sending messages across the same open communications channel.

Large values of a , b , and p are needed to make this method secure, since the security in this method relies in the unfeasibility of determining the secret values a or b from the public values A or B . The problem such a computer needs to solve to break this method is called the discrete logarithm problem.

Of all the numbers used, g is only one that is not required to be large. It can actually be a small prime number like 2 or 3 because the primitive roots are usually quite numerous.

Table 4. Diffie-Hellman protocol original implementation using encryption mathematics.

Alice				Bob		
Secret	Public	Calculates	Sends	Calculates	Public	Secret
a	p, g		$p, g \rightarrow$			b
a	p, g, A	$g^a \bmod p = A$	$A \rightarrow$		p, g	b
a	p, g, A		$\leftarrow B$	$g^b \bmod p = B$	p, g, A, B	b
a, s	p, g, A, B	$B^a \bmod p = s$		$A^b \bmod p = s$	p, g, A, B	b, s

If an intruder were to capture every value that was transmitted, he would only have the values p , g , A and B . Since s , to be calculated, requires the knowledge of both private values a and b and given that they are unfeasible to be determined from the public values A and B (with a big enough p), then s remains secret.

This protocol is, however, vulnerable to man-in-the-middle (MITM) attacks since neither Alice nor Bob are authenticated and an intruder could intercept the communication and agree two different keys (one for Alice and another for Bob) and then relay the communication while being able to decrypt it.

3.3.2.1.3 Asymmetric Keys

In this section we will present asymmetric keys and how they can be used. Public (or asymmetric) key cryptography involves using asymmetric key pairs which have a public part that can be known by anyone and a private part that only the owner of the asymmetric key pair should possess. An algorithm that allows to generate pairs of asymmetric keys is the RSA algorithm, created by Ronald (R)ivest, Adi (S)hamir e Leonard (A)dleman, which became one of the most successful implementations.

Asymmetric keys can be used to encrypt a piece of data using one part of the asymmetric key and the result can only be decrypted using the other part. This property, together with the fact that one part of the key is made public and the other isn't, leads to two different scenarios: Encryption using the public part of the key and encryption using the private part of the key.

When some data is encrypted using the public part of the key, then it must be decrypted using the private part. This means that only the owner of the private part of the key will be able to decrypt the data, meaning that the data is kept secret to other people.

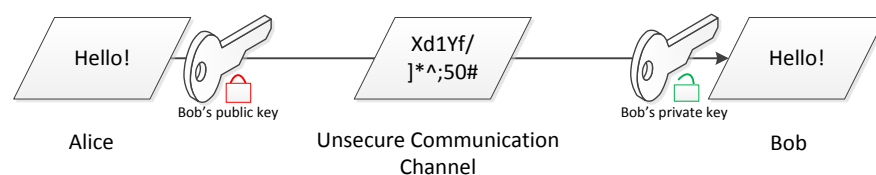


Figure 18. Message encryption over an unsecure network using an asymmetric key pair.

On the other hand, if the data is encrypted using the private part of the key then only the owner of the key could have encrypted it, assuming the private part of the key is kept secret. Because the data can be decrypted with the public part of the key, which is known to anyone, the data is not actually secret, but if the public part of the key decrypts the data then the person who decrypts it can assume it was the owner of the private key that encrypted it. This can be used as a digital signature. This last scenario has the drawback of actually sending the same message twice, which increases the overhead in the communication. To address this situation, a hash of the message is signed instead of the actual message. This means that the receptor must hash the message received using the same hashing algorithm and compare it with the decrypted signature. Since hashes are usually much smaller than the message hashed the overhead is decreased, but messages that generate the same hash (a collision) will

carry the same signature, meaning that the actual signed message could be swapped by one such collision by an attacker, so a good hash algorithm where collisions for a given message are hard to find should be used.

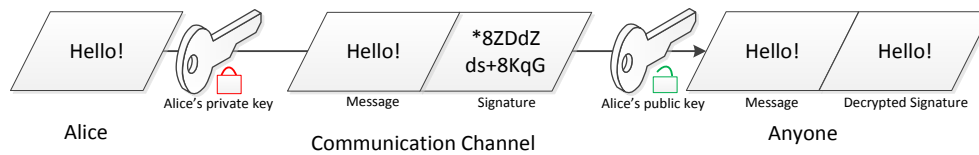


Figure 19. Message signing using an asymmetric key pair.

It is important to note that these cases are not mutually exclusive in the sense that both can be used on the same message. By signing the data using one's private key and then encrypting everything with the receiver's public key will mean that only the receiver will be able to decrypt the data and the signature and use the sender's public key to verify the signature of the data, allowing to authenticate the sender and preventing anyone else from reading the data.

3.3.2.1.4 Public Key Certificates

In this section we will present the notion of public key certificates and how can be used for data encryption. The distribution of keys becomes easier using public key cryptography. Each node is responsible for knowing its own private key and all the public keys can be accessed from one place. But there is a problem as well. The public keys can be listed in a journal or stored in a directory service, but if an intruder changes the public key associated with someone, the intruder can then impersonate them, because no one other than the owner of the private key can verify that the public key is incorrect. Public key certificates are an electronic document that can solve this problem to some degree. It uses a digital signature to associate a public key (from an asymmetric key pair) to an entity, which can contain information such as the name of a person or organization, the address and the email address. Public key certificates can be used to verify that a public key belongs to an entity, by having a trusted entity (Certificate Authority) to sign the public key and its owner information, as shown in section 3.3.2.1.3. When a client wants to verify the entity that signed some data, the server will send its own public key certificate with the public key that can decrypt the signature of the data it sent. The client, by having the trusted certificate authority's public key preconfigured, can verify the signature on the public key certificate received from the server, allowing the client to have some confidence in the identity of the server. The level of confidence depends on the level of trust the client has on the certificate authority in its ability to sign certificates that associate the public keys to their legitimate owners.

If the private key associated with the public key in a certificate is compromised, then the entity that owns the public key certificate must revoke it so that it no longer remains valid. This mechanism uses certificate revocation lists (CRL), also signed by the certificate authority, that the client can check to determine if a given public key certificate is still valid or not. A public key certificate is only valid if it

has a valid certificate authority signature, if it is still not expired and if it isn't listed in the most recent CRL from the certificate authority. A public key certificate expires after a given amount of time, typically a year.

An alternative to the CRL was designed to keep the servers from having to keep CRLs updated and to parse them, called Online Certificate Status Protocol (OCSP)(IETF, n.d.-a). This protocol, compared to CRLs, carries less information in the responses, so it can use the network and the OCSP client resources more efficiently.

3.3.2.2 SSL/TLS

In this section we will present Secure Sockets Layer (SSL)(IETF, n.d.-b) and its successor, Transport Layer Security (TLS)(IETF, n.d.-c). They are cryptographic protocols that provide communication security over the internet and allow client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering and message forgery.

SSL and TLS are protocols that can use public key certificates to authenticate a server and, optionally, the client. It also encrypts the communication channel using a set of keys that are calculated during the initial handshake process, where information required to establish a secure communication channel is exchanged, such as the protocol version, ciphers, public key certificates and other values that may be required by the cipher.

This section is divided as follows: Section 3.3.2.2.1 will present the basic SSL/TLS protocol and section 3.3.2.2.2 will present the anonymous version of it.

3.3.2.2.1 Basic protocol

In this section we will present the SSL/TLS basic protocol. The SSL/TLS protocol uses the Transmission Control Protocol (Kaufman, 2002b), a reliable transport layer that provides a reliable, ordered and error-checked stream of octets (a unit of information consisting of 8 bits). It also provides congestion and flow control, which prevents the sender from congesting the network or overflowing the receiver with information, respectively. SSL/TLS puts together these octets into records, which contains headers and cryptographic protection to further provide a stream with integrity and encrypted protection on top of the reliable delivery of data.

There are four types of records: handshake, change cipher spec (which are used during handshake but have been made into a different type of record), data and alerts (to alert about errors or notifications of connection termination).

In the basic protocol, the client (Alice) connects to the server (Bob), whom sends its public key certificate. Alice verifies the certificate, obtains the public key from the certificate, generates a random number S that will be used to derive the session keys and sends it to Bob encrypted with his public key. The session keys are calculated and then used to guarantee the integrity and encryption of the data.

Figure 20 shows a simplified form of the protocol, which we will discuss in more detail.

In the first message, Alice indicates that she wants to communicate with Bob using a ClientHello message, but doesn't say who she is. Provides Bob with a list of ciphers she supports and a random number R_{Alice} that will be used to derive the session keys from S . The second message, a ServerHello sent by Bob, contains the cipher that he chose from the list received that he supports and a random number R_{Bob} which will serve the same purpose as R_{Alice} . Bob then sends a Certificate message with his public key certificate (if the chosen cipher requires it) followed by a ServerHelloDone to finish the handshake negotiation. Next, Alice chooses the random number S (known as the pre-master secret) and sends it, encrypted with Bob's public key, in a ClientKeyExchange message.

Now both parties use this pre-master secret and the random values to generate the master secret. All the other required keys for this connection are derived from this master secret and the random values R_{Alice} and R_{Bob} using a pseudorandom function.

Alice then sends a ChangeCipherSpec record message to indicate that future messages will be encrypted and then an encrypted Finished message indicating that the client's handshake has finished, together with an hash and a message authentication code (MAC, i.e. a hash function that takes a cryptographic key as input) of the previous handshake messages. Bob decrypts the message and verifies both the hash and the MAC. Finally, Bob also sends a ChangeCipherSpec message and an equivalent Finished message indicating that its handshake has also finished. Alice performs the same decryption and verification. From this point on the handshake is completed and the SSL/TLS protocol moves to the application phase, allowing Alice to communicate with Bob using integrity-protection and data encryption.

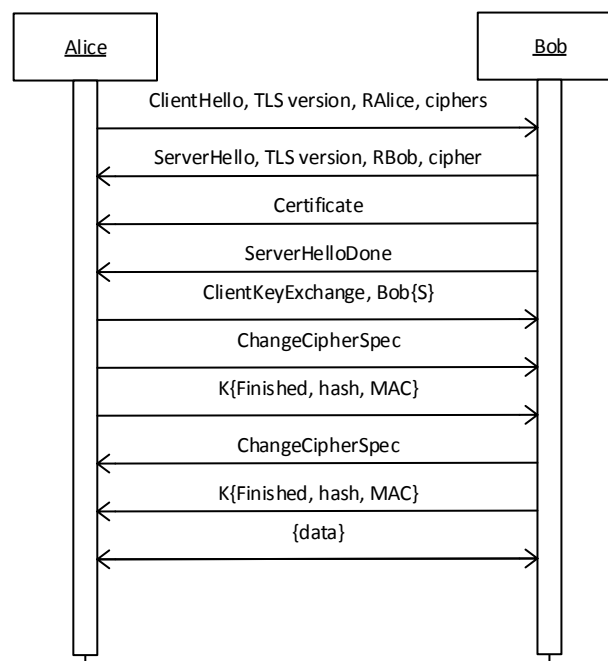


Figure 20. Basic TLS protocol using server certificate.

Note that only Bob was authenticated in this exchange. The protocol could be used to authenticate both Bob and Alice, e.g. if Alice used a certificate of her own, but the most common way to authenticate the client Alice is for her to send the username and password through the encrypted communication channel to the server Bob once it is created.

3.3.2.2.2 Anonymous SSL/TLS

In this section we will present the anonymous version of the SSL/TLS protocol and what differs from the basic version. The SSL/TLS protocol can use many combinations of authentication mechanisms and key exchange protocols, among other aspects like the encryption mode, mechanism and MAC algorithm.

Cipher suite's names follow a common format, e.g. in *SSL_RSA_WITH_DES40_CBC_SHA*: SSL means SSLv3. Other values are SSL2 for SSLv2 or TLS for TLS version 1.2. RSA is the algorithm used to encrypt the material used to generate the session keys that are sent over the network with asymmetric keys. The algorithm uses certificates with RSA keys. WITH does not have any special meaning, it serves no other purpose than increasing the suite's name length. DES40 means that DES with 40 bit keys will be using as the encryption algorithm. CBC stands for cipher block chaining and it's the encryption mode(Kaufman, 2002c). SHA is the algorithm used for generating MAC values.

The most interesting section for the purpose of implementing S-DRACA is the key exchange algorithm. The basic protocol presented previously would use RSA as the key exchange algorithm since the server Bob provided its public key certificate, which is used for asymmetric key encryption.

The usage of certificates means that the owner of the server Bob and each client must trust the CA that signed the server certificate to never provide to an entity a certificate declaring that they are another. If that is possible then there's no guarantee that a server certificate actually authenticates the server, it could be other entity that was able to get the CA to sign the impersonating certificate.

To disable the usage of certificates, the algorithm used to securely exchange the material that is used to generate the session keys must be changed. One example is DH_anon, which uses Diffie-Hellman to exchange the material needed (the agreed key becomes the pre-master key). Since no entity is authenticated using Diffie-Hellman it is considered that the communication is anonymous.

Because the Diffie-Hellman protocol requires both parties to send each other material to be able to arrive at the agreed key, a change to the previous basic protocol is required: the server no longer sends its public key certificate and sends a ServerKeyExchange message instead with the Diffie-Hellman public parameters.

3.3.3 Authentication

In this section we will present authentication and other related protocols. Authentication is the process of reliably confirming the identity of something or someone who identifies itself. There are three categories of mechanisms in which the entity can be authenticated based on the type of proof (or factor of authentication) needed, which can be:

- Something that the user is, e.g. a fingerprint or a retinal pattern;
- Something that the user has, e.g. a smartcard or a cell phone;
- Something that the user knows, e.g. a password or an answer to a particular question.

We will analyze some mechanisms and protocols for entity authentication that were used in this dissertation. SSL/TLS, presented in the last section due to its encryption capabilities, can also authenticate the client and the server.

This section is divided as follows: Section 3.3.3.1 presents the basic password authentication protocol, section 3.3.3.2 presents the challenge-response authentication mechanism, section 3.3.3.3 presents how authentication can be obtained using encrypted key exchange and section 3.3.3.4 presents smart cards and how they can be used for authentication.

3.3.3.1 Password Authentication Protocol

In this section we present the password authentication protocol, which is a very simple protocol (see Table 5), where the client (Alice) sends her username and password in clear text on the network, and the server Bob just acknowledges the authentication if the password matches the username or he doesn't acknowledge it if it doesn't.

This protocol requires that both Alice and Bob share a common secret (the password). It doesn't require anything more complex than an equality check to authenticate the client and the communication is minimal, which means that this protocol has a low overhead, but it is only suitable in constrained environments where eavesdropping in the network is not an issue, or else the password is very easily stolen.

Table 5. Password authentication protocol.

#	Alice	Network	Bob
1	username, password	→ username, password	username, password
2	username, password	← ack/nack	username, password

3.3.3.2 Challenge-Response Authentication Mechanism

In this section we present the challenge-response authentication mechanism, which is a family of protocols where one party presents a question, the "challenge", which can be anything, and the other party provides the answer, the "response". If the response is the expected one to the challenge, then it is authenticated.

One simple example of a mutual authentication scheme (where both the client and the server are authenticated) is shown in Table 6. The challenge in this example is a random value, to which the correct response is the hash of the challenge sent to the other party with the received challenge and the password.

First the client (Alice) identifies herself and sends a random challenge C_1 to the server (Bob). Bob then generates a random challenge C_2 and sends the response to Alice's challenge, together with

his challenge, in the second message. In the third message Alice sends her response to Bob's challenge if Bob's response matches the expected value, who replies with an acknowledge message if Alice's response matches the expected value or with a not acknowledge otherwise.

Table 6. Challenge-Response authentication mechanism protocol.

#	Alice	Network	Bob
1	username, password, $C_1 = \text{random}()$	\rightarrow username, C_1	username, password
2	username, password, C_1	$\leftarrow C_2, \text{hash}(C_2, C_1, \text{password})$	username, password, $C_1, C_2 = \text{random}()$
3	username, password, C_1, C_2	$\rightarrow \text{hash}(C_1, C_2, \text{password})$	username, password, C_1, C_2
4	username, password, C_1, C_2	$\leftarrow \text{ack/nack}$	username, password, C_1, C_2

Note that Alice's password is never transmitted in clear text on the network, but it can be easily guessed from the responses if it is a weak password (small with low complexity like only using lower case characters).

3.3.3.3 Encrypted Key Exchange

In this section we present the Encrypted Key Exchange (EKE)(S. Bellovin, 1992)(S. M. Bellovin, n.d.), which is a family of protocols that provide key agreement mechanisms that are password-authenticated. The protocols of this family take a shared password and use it to generate a shared key. The basic form of EKE consists of one party of the communication to encrypt an ephemeral (one-time) public key and sending it to the other, who decrypts it and uses it to negotiate a shared key.

The Simple Password Exponential Key Exchange (SPEKE)(Jablon, n.d.) is a cryptographic method for password-authenticated key agreement. This variation of the Diffie-Hellman key exchange protocol derives the generator g used from the shared password. Since the generator g can only be obtained from the password, only the entities that know this password can calculate it, hence Alice and Bob will arrive at the same shared secret only if they have used the same password. Once they have the shared secret they can prove to each other that they know the password by confirming that each arrived at the same shared secret, e.g. by sending an asymmetric hash of it concatenated with different known strings.

An augmented version of this family of protocols, called Augmented Encrypted Key Exchange (S. Bellovin, 1993), describes a concept where it is ensured that password verification data stolen from a server cannot be used directly by an intruder to impersonate as a client. The Secure Remote Password (SRP)(Wu, 1998) is one such protocol, therefore it prevents someone who was able to steal the server database from being able to impersonate the user. The protocol creates a shared private key between two parties similarly to Diffie-Hellman, but it is more complex as it uses a password verifier in

the server side instead of the client's own password that, if stolen, cannot be used directly to allow the intruder to impersonate the user.

In short, here are some of the properties that make SRP a strong authentication protocol. 1) SRP is safe against replay attacks (explained later in section 3.3.4.1). None of the data transmitted during authentication can be re-used to authenticate to a server using SRP. 2) SRP is safe against eavesdroppers. The password is never transmitted, either in clear text or encrypted. 3) SRP supports mutual authentication. 4) SRP transmits a session key in the process of authentication. This key can be used to encrypt the user's session and protect it from both eavesdropping and malicious active attack. 5) SRP is safe from off-line dictionary attacks against the transmitted messages. The messages exchanged over the network are insufficient to verify a guess of a user's password. 6) SRP provides perfect forward secrecy. A compromised password does not enable an intruder to decrypt past sessions and a compromised session key will not enable an intruder to find out the user's password. 7) SRP can tolerate a compromise of the verifier database on the host. Even though such a compromise may enable some attacks against the system (dictionary attack (explained later in section 3.3.4.3), host impersonation, etc.), it may not be catastrophic, because the password verifiers can only be used for validation of a user's password (i.e. they are not the plaintext-equivalent to the actual passwords). Therefore, they cannot be used by an intruder to gain direct access to a server.

The last three issues are difficult constraints to satisfy. If one considers only protocols that resist dictionary attacks, one is left with the EKE family of protocols discussed in this section and a few other public-key assisted protocols such as SSL/TLS. If one also requires perfect forward secrecy, that leaves only the strongest of the EKE family protocols, like DH-EKE and SPEKE. SRP is able to satisfy both those constraints and the final requirement for non-plaintext equivalence.

3.3.3.4 Smart card

In this section we present smart cards, which are pocket-sized cards that have integrated circuits embedded into them and can be used for many reasons, including entity identification, authentication, data storage and processing. An example of such cards are the cellphone's SIM card which can perform the processing required for the cellphone to be able to use the radio network.

In terms of the type of authentication factor, they are something that the user has and must use to authenticate with the systems that require them. These cards usually have to be unlocked with a PIN to be used, meaning that these also require something the user knows.

When two factors of authentication from different categories are required to authenticate an entity then it is known as a two factor authentication (TFA). Because this type of authentication requires more than one type of proof they are usually safer and the probability of a successful impersonation is decreased.

Smart cards that can authenticate and prove the identity of an entity are our main focus. These usually provide a public key infrastructure (i.e. a set of hardware, software, people, policies and procedures needed to create, manage, distribute, use, store and revoke digital certificates), which

stores digital certificates issues by the public key infrastructure provider together with other relevant information. Note that, by design, it isn't usually possible to extract the asymmetric private key from the cards and only the card itself can use it to sign data.

The smart card used in this dissertation was the Portuguese Citizen Card (CC) ("Cartão de Cidadão," n.d.). It will be used by our system as part of its TFA.

The card itself publicly provides access to information such as the card holder's name, the several identification numbers associated with the card owner (e.g. the social security number, the identification number, etc.) and the public key certificates, i.e. the signature and the authentication digital certificates. After unlocking the card we have access to more personal information such as the current address of the card's owner and the ability to request the usage of the private keys associated with the public keys in the digital certificates.

The digital certificates in every CC are signed by a Portuguese government certificate, which changes every year. Those certificates are signed by a stable certificate that also belongs to the Portuguese government. Since the government's public key certificates are available online we can download them to authenticate a user, for which we have to request its public key certificate. Then we use another authentication mechanism, like the challenge-response, where the challenge must be signed using the private key associated with the public key in the received certificate. If the signature is valid (i.e. decrypting the signature using the public key produces the original challenge or a hash of it, depending on the implementation) and the public key certificate states that the user is, in fact, who he claims to be, then all that remains is to determine that the public key certificate is actually signed by the government's certificate (our trust anchor). If the signature of the public key certificate is validated using the public key certificate of the government's certificate AND none of the certificates have been revoked, then we can assume the user is who he claims to be and authenticate him.

To simplify the revocation verification process and prevent the server from having to parse the CRLs the Portuguese government has OCSP responders publicly available.

3.3.4 Security attacks

In this section we will discuss some of the possible attacks that can be performed against computer systems. After discussing the various forms of user (and optionally the server) authentication, we analyze in this section the possible attacks that can be performed against such mechanisms and show how to prevent them if possible. It is important to understand the vulnerabilities that can affect each authentication method to safely decide which mechanism is the most appropriate for a given scenario.

This section is divided as follows: Section 3.3.4.1 presents the replay attack, section 3.3.4.2 presents the exhaustive key search attack, section 3.3.4.3 presents the dictionary attack, section 3.3.4.4 presents the pre-computed dictionary attack, section 3.3.4.5 presents the reflection attack and section 3.3.4.6 presents the man-in-the-middle attack.

3.3.4.1 Replay attack

In this section we present the replay attack, which is a simple attack that targets systems where the client's password (or a hash of it) is sent through the network to authenticate the client. The attack consists of an eavesdropper that captures an authentication interaction between a client and a server, collecting the client's (hashed) password in the process. Then the attacker can authenticate with the server impersonating the previous client by providing its (hashed) password.

To prevent this kind of attacks a more sophisticated authentication protocol can be used, such as Challenge-Response, EKE or SSL. One-time passwords are another alternative, which are passwords that are sent to the server through the network but that can only be used once.

3.3.4.2 Exhaustive key search

In this section we present the exhaustive key search attack, sometimes called a brute-force attack, which is an attack that can be used against any encrypted data given that the attacker has enough information to break the encryption (e.g. the attacker has to know something about the data that is encrypted, such as its structure, so that he can validate a decryption attempt).

This attack consists of checking all possible keys or passwords that are valid until the correct one is found. This process can be done online or offline. The online attack tries to guess the password by attempting to authenticate with the server, for example. A way to prevent online attacks is to limit the number of tries a client can perform, introducing delays between consecutive attempts, using CAPTCHAs, etc. In the offline attack the attacker has access to encrypted data that he can try to decrypt without the risk of discovery or interference.

This attack can be fast when checking small passwords, but for longer passwords it becomes unfeasible and a dictionary attack is performed instead (discussed in section 3.3.4.3). In the worst case it would require traversing the entire search space.

A password with 6 letters has a search space of just over three hundred million passwords, which can be easily tested with some computational power. Using bigger passwords, especially with numeric digits, upper case, lower case and special characters, the search space becomes too big to be feasible. Human beings, the main users of passwords, do not usually use random passwords but known words with few variations such as adding a digit to the end, which leads to more effective attacks such as the dictionary attack discussed next.

3.3.4.3 Dictionary attack

In this section we present the dictionary attack, which is similar to the exhaustive key search. This attack consists of trying to guess the password or key to decrypt data. However, this attack does not attempt to go through the entire search space. Instead, it tries all the words defined in a list called a dictionary. The words in the list are the most likely words to be used for a password, typically from an actual dictionary, since passwords are used by people and it's easier to remember an actual word

instead of random characters. The list used can also test variations of words, e.g. by appending a digit to the end of a word, which makes those variations easy to guess. However, if the word has a random character in the middle it becomes difficult to guess. This attack reduces the time it takes to guess the password, but unlike the exhaustive key search it is not guaranteed to succeed.

3.3.4.4 Pre-computed dictionary attack

In this section we present the pre-computed dictionary attack, also known as a rainbow table attack, which allows achieving a time-space tradeoff by storing a password in a map and indexing them using their hash value. If the hash of the user's password is intercepted in any way then the attacker can use this map to quickly determine the original password of the user. The creation of the map requires some preparation, since the hash of every word in the list needs to be calculated and stored, but it only needs to be created once, since the hash values never change for a particular hashing algorithm.

These types of attacks can be thwarted with the use of a salt. A salt is random data that is used as an additional input to the hash function. Using different salts for the same password will result in a different hash value, so provided that the salt is large enough it is impossible to use a pre-computed dictionary attack because the map would have to contain an entry for each password-salt pair.

3.3.4.5 Reflection attack

In this section we present the reflection attack. Regarding the authentication mechanisms, an attack known as the reflection attack can be performed against some implementations of the challenge-response mutual authentication mechanism shown in section 3.3.3.2. The idea is to use the other party to provide the answer to their own challenge.

Assuming the target is Bob and the attacker is Alice, the general attack goes as follows:

1. Alice connects to Bob and sends a challenge to authenticate him;
2. Bob sends its challenge C_b to Alice to attempt to authenticate her and the response to Alice's challenge;
3. Alice starts another connection to Bob and sends the challenge C_b previously obtained from Bob;
4. Bob responds with its own challenge and the response R_b to the challenge C_b ;
5. Alice goes back to the first connection and sends Bob the response R_b as the answer to the challenge C_b and finishes the authentication.

As we can see the target ends up sending the response to the challenge that was sent by him on a separate connection. This leaves Alice with one fully authenticated channel connection (the second connection is simply dropped).

There are several ways to protect a system against this type of attack: the challenges used by the client and the server can be exclusive, e.g. a client's challenge might have to be even or have the client's identifier appended and the server's challenge has to be odd or have the server's identifier added; the

system can require a client to answer its challenge before the server responds, preventing an attacker from acquiring responses to challenges without being authenticated; or require the key used to calculate the responses from the challenges to be different on both parties.

3.3.4.6 Man-in-the-middle attack

In this section we present the man-in-the-middle attack. MITM attacks are a type of attack where the attacker takes an active stance in the communication between two parties in which the attacker makes independent connections between each of the victims, making them think that they are talking directly to one another in a private communication channel (see Figure 21).



Figure 21. MITM attack exemplification where Eve is the attacker.

Most cryptographic protocols use some sort of endpoint authentication to prevent this type of attacks specifically. As shown previously, SSL can authenticate one or both parties using a trusted certification authority and key agreement protocols of the EKE family allows for two parties to agree on a secret key based on a shared password while preventing this type of attacks.

To exemplify let's go through a scenario where Alice wants to communicate with Bob while Eve wants to listen to the conversation and send a false message to Bob (optional step).

Table 7 illustrates the messages sent by each party in the communication. Alice starts by requesting Bob's public key to encrypt the message (1). Eve intercepts the message and forwards it to Bob. Then Bob replies with this public key, which is intercepted by Eve and stored, sending Eve's public key instead (2). Finally, Alice believing to have Bob's public key encrypts a message and sends it. Eve intercepts the message, decrypts and modifies it, encrypts it with Bob's public key and sends it to him (3).

Table 7. MITM attack scenario.

#	Alice	Eve	Bob
1	→ Bob, it's Alice. What's your key?	→ Bob, it's Alice. What's your key?	
2		← [Eve's Key].	← [Bob's Key].
3	→ Meet me at seven! [encrypted with Eve's key]	→ Meet me at eight! [encrypted with Bob's key]	

Protocols that do not authenticate any of the parties involved in the communication are vulnerable to this type of attack. One example is the Diffie-Hellman key exchange protocol, where a

MITM attack can be used to intercept the agreed keys. However, variants to the protocol in the EKE family, such as DH-EKE, solve this problem by authenticating the parties in various ways.

3.4 Summary

In this section we discussed the background of some technologies which were used in S-DRACA, in an attempt to enable the reader to be able to understand the work without having to rely on outside sources of information. This included some service coordination notions that will be used when explaining the implementation of the sequences of CRUD expressions, some essential Java functionalities and an overall view of some computer security aspects such as cryptography and authentication of users since we will be implementing mechanisms to provide these functionalities.

4 Secure, dynamic and distributed RBAC Architecture

In this section we describe S-DRACA, the secure, dynamic and distributed access control architecture which is built upon previous research [14][56][57]. We will present its architecture, how it enforces the access control policies and their respective sequences. We will also present the security layer to secure S-DRACA, a performance evaluation and a proof of concept.

S-DRACA is an access control architecture that aims to provide an easy interaction between relational applications and relational databases, similar to the interaction offered by other API's like JDBC(Oracle, 1997b), ADO.NET(Microsoft, n.d.), JPA(Oracle, 2006), LINQ(Microsoft, 2007) or Hibernate(JBOSS, 2001), while controlling the access to sensitive information. S-DRACA, like JPA and Hibernate, builds a framework in compile-time that users can use to access the sensitive information removing the need for them to master the database schemas and the enforced RBAC access policies. It does, however, enforce changes made to the RBAC policies in runtime. This means that an application can change its behavior when a change is made, avoiding security exceptions and the need to change the application. Furthermore, the generated framework is easily updatable during the compilation of the application, which means that if the application is trying to access information it does not have authorization to a compilation error is generated, instead of a runtime exception in the case of the existing solutions that can only be found using tests or during production, decreasing the application's time to market and increasing user satisfaction.

This section is divided as follows: Section 4.1 will present the overall architecture of S-DRACA, including an overview of DACA since it was used as the starting point, section 4.2 will present the changes made to the access control policies' layer, section 4.3 will explain the new sequence controller, section 4.4 will present the overall system security concerns and mechanisms implemented, section 4.5 will show a study on the overall performance of the solution and section 4.6 will present a proof of concept that uses S-DRACA to access sensitive data with dynamic modification of policies.

4.1 S-DRACA overall architecture

In this section we will present the architecture stack, the overall architecture of DACA and S-DRACA and finally the process of implementing the RBAC policies.

We start by presenting the architecture stack, shown in Figure 22. The application layer interacts with the Sequence Controller layer, which provides the application with a set of services that are regulated by the access control policies and provides a finer control over how the access control mechanisms are used. The data access itself is done by the access control mechanisms, which interacts with the security (SSL/TLS and Authentication) layer to provide increased security. Finally the data layer contains the sensitive information we want to provide access to.

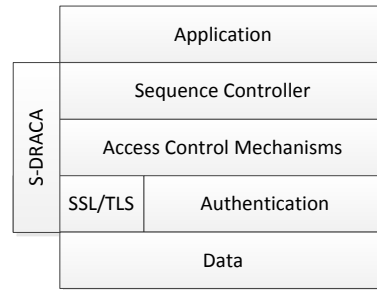


Figure 22. S-DRACA stack.

S-DRACA can be divided in two different sections: the client side, where the users' access to the sensitive data is controlled using the generated architectures' access control mechanisms, and the server side, where the RBAC policies that regulate the mechanisms in the client side are stored. The security layer stays between them to provide secure communication and authentication.

Unlike the PEP-PDP architecture, where each user request to the PEP implies a verification with the PDP, or the SAAM architecture, shown in Figure 5, where such a request to the PDP is still made if the SDP cannot authorize the request, S-DRACA always decides if the user's request is allowed or not in the client side. This has several advantages over the alternatives, of which we emphasize: the decision to allow a request or not is made at the client side, which implies that the time to authorize a request is lower, the user experiences less wait time and the requests are no longer sent to a PDP in the server side, meaning that a central point of failure and a possible performance bottleneck is removed. It has some disadvantages, however: the initialization time is greater, since the client has to receive the permissions a user has, given its role. This problem is minimized if the user uses the same session for a long time, because the initialization is only required to be done once per session. The security management also has to be carefully designed, given that a request is authorized at the client side. It should be made impossible to bypass S-DRACA using any means.

This section is divided as follows: Section 4.1.1 will present an overview of DACA, used as the starting point, Section 4.1.2 will present an overview of the S-DRACA and how it differs from DACA. Section 4.1.3 will detail the S-DRACA's architecture which was worked on, section 4.1.4 will explain how the interfaces the programmers use to access the data are generated and implemented and section 4.1.5 briefly summarizes the contents presented. The layers that were developed will follow afterwards.

4.1.1 DACA overview

In this section we present a brief overview of DACA, which was used as a starting point to provide S-DRACA with the capabilities to adjust to policy changes and relieve programmers from having to master the database schema and the access control policies.

Figure 23 shows a block diagram of DACA. It is comprised of four main components: the Policy Manager, the Policy Extractor, the Business Manager and the access mechanisms.

The access control policies, which regulate what a client application can do with the sensitive data, are available to the Policy Manager in a database called Policy Server. It can send the access control policies to the client side components and notify the clients when these policies change. To do this, it keeps a list of active clients with their IP address and port number.

The Policy Extractor is responsible for obtaining the access control policies from the Policy Manager and generates interfaces that make the client application aware of the access mechanisms, which the client application can use to access the data in the database. In DACA, the Policy Extractor is a separate application, which bundles the generated interfaces in a Jar file that can be used in client applications.

When the client application wants to access the sensitive data using the access mechanisms, it requests their instantiation to the Business Manager. The Business Manager also receives the access control policies from the Policy Manager in order to implement the access mechanisms used by the client application.

The access mechanisms are where the enforcement of the access control policies is made, providing the client application with only the authorized operations. These are only implemented and loaded in runtime, to prevent the manipulation of the implemented source code.

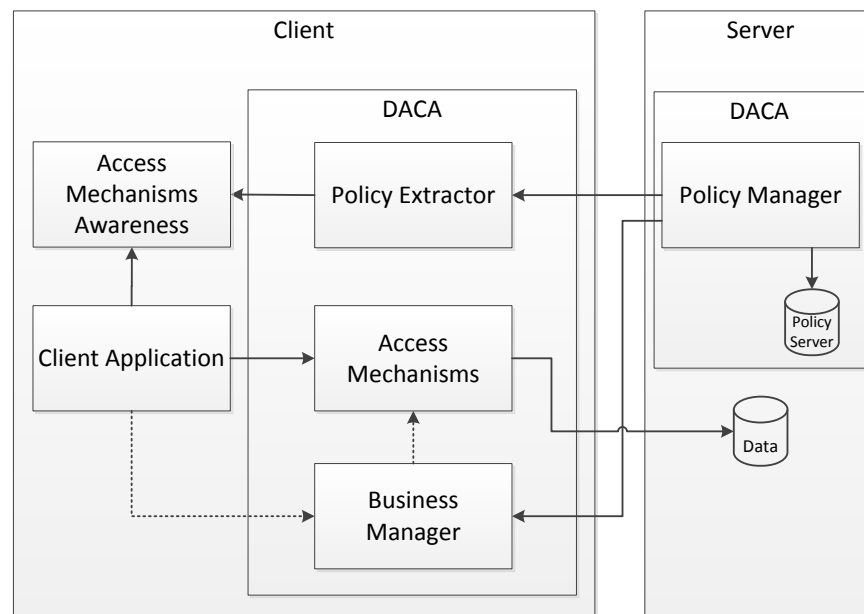


Figure 23. DACA architecture block diagram.

4.1.2 S-DRACA overview

In this section we will present a brief overview of S-DRACA and how it evolved from DACA. Looking at Figure 24 and comparing it with Figure 23 we can see that the main four components, i.e. the policy manager, the policy extractor, the business manager and the access mechanisms are still present.

However, these components are not exactly the same. Although they provide the same functionalities, they have been given new ones.

The Policy Manager, additionally to sending the access control policies back to the client side components and notify the clients when these policies change it now performs the authentication of clients and can establish encrypted communication channel to protect the sensitive data from intruders listening to the network.

The Policy Extractor has been re-implemented to be directly integrated in the client application using java annotations, but its main functionality of obtaining the access control policies from the Policy Manager and generating the awareness components for the access mechanisms remains the same.

The Business Manager still receives the access control policies from the Policy Manager in order to implement the access mechanisms used by the client application, but it now also manages the enforcement of the sequences of CRUD expressions. Hence, it keeps track of the state of the sequences being executed, authorize their execution and adjust them dynamically when the defined sequences are changed in the Policy Server.

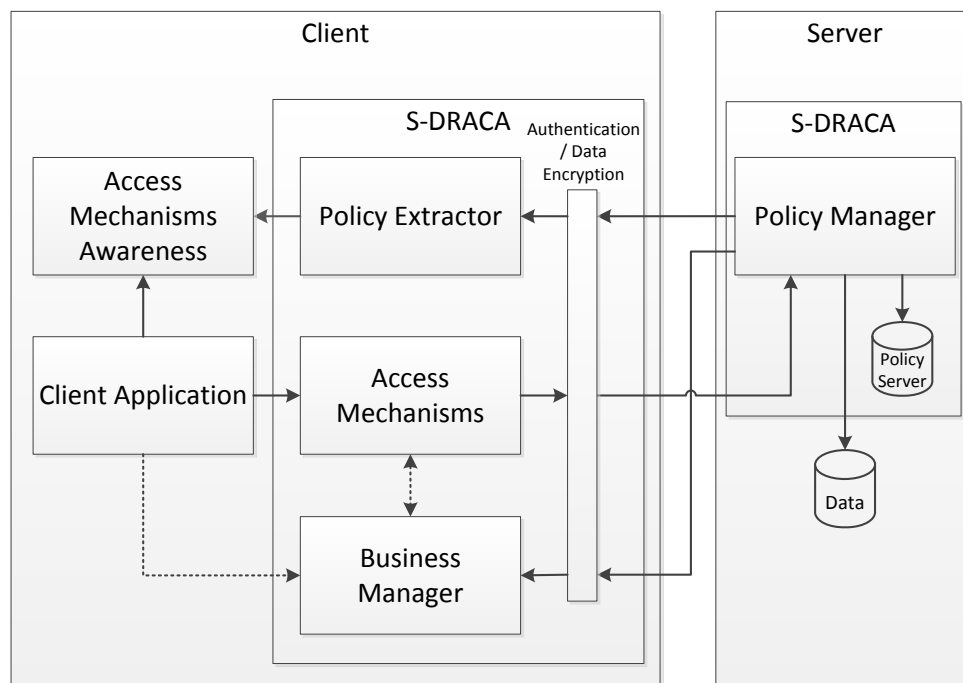


Figure 24. S-DRACA architecture block diagram.

The access mechanisms remain where the enforcement of the access control policies is made, providing the client application with only the authorized operations, but now they also allow the client application to request the next CRUD expression to be used in a sequence. Since the Business Manager is the component responsible for authorizing these operations, the access mechanisms now also make requests to it.

In addition to these components a new one has been created, named “Authentication / Data Encryption” in the previous figure. This component mediates the communication between the client

side S-DRACA components and the Policy Manager in the server side, providing authentication and data encryption mechanisms. Additionally, DACA would send the CRUD expressions to the client side so that the access mechanisms could access the data. Because of that, the CRUD expressions are now pushed to the server side to prevent intruders in the client's side from being able to access and modify them. This component is further explained in section 4.4.

4.1.3 S-DRACA's architecture

In this section we will present the S-DRACA architecture for the enforcement of the extended RBAC policies. Figure 25 shows the S-DRACA's architecture that evolved from DACA with the exception of the authentication and data encryption component which mediates the communication between the client and the server. Furthermore, there is a software architectural model, presented in section 4.2.3, used to build entities known as Business Schemas, which are the generated access mechanisms.

The Business Schema is the most important entity in DACA, and therefore in S-DRACA as well, since they allow the clients to access the data in the database while providing only the authorized operations, through a CLI that abstracts the JDBC connection and statement object's interfaces. Note that they are also implemented at runtime and they possess all the information required to enforce the defined access control policies at the client side, creating a distributed solution. To build these Business Schemas we make use of a small set of interfaces that define protocols for basic operations, such as execute a CRUD expression, update a row in a LDS, etc. Then, depending on whether these operations are authorized or not, these smaller interfaces are implemented by a Business Schema or not. This way, only authorized operations are implemented and available to the client, even using functionalities like Java Reflection. This process is further explained in section 4.2.

The overall architecture is very similar, but a new component has been added to the Business Manager: the sequence controller. The sequence controller is the component responsible for deciding if an action being executed is in a valid context, i.e. the order in which the previous actions are been executed, together with the new action being requested, remains a valid sequence. The sequence controller is further explained in section 4.3. Another major difference from DACA, is that the Business Schemas implementation no longer connects directly to the database with the sensitive data, reason being that having the client possess the credentials to the database would be a security vulnerability. With them, the client would be able to completely bypass the S-DRACA access control enforcement layer by connecting directly to the database.

Figure 25 also shows the normal usage of S-DRACA. First, a policy extractor (an external tool) connects to the Policy Manager and requests the defined policies (1). From these policies and the architectural model shown in, the policy extractor is capable of generating the interfaces for the Business Schemas that the application developers can use (2). Then, at runtime, the Business Manager requests, on behalf of the application, the policies from the Policy Manager (3) and implements the interfaces generated previously by the Policy Extractor (4). Modifying the interfaces does not allow a user to use some unauthorized operation, because the implementation of the Business Schemas does

not take into account the interfaces generated by the Policy Extractor. If they do not match, a runtime exception occurs when the Business Manager tries to compile the generated implementations at runtime. Then the application can instantiate and use the Business Schemas it is allowed to use (5). If any operation or instantiation is not in the defined allowed sequence, the Sequence Controller will generate a runtime exception. Finally, when an operation is requested to a Business Schema, it uses a connection to the Policy Manager to talk to the database (6). The Policy Manager relays the communication between them.

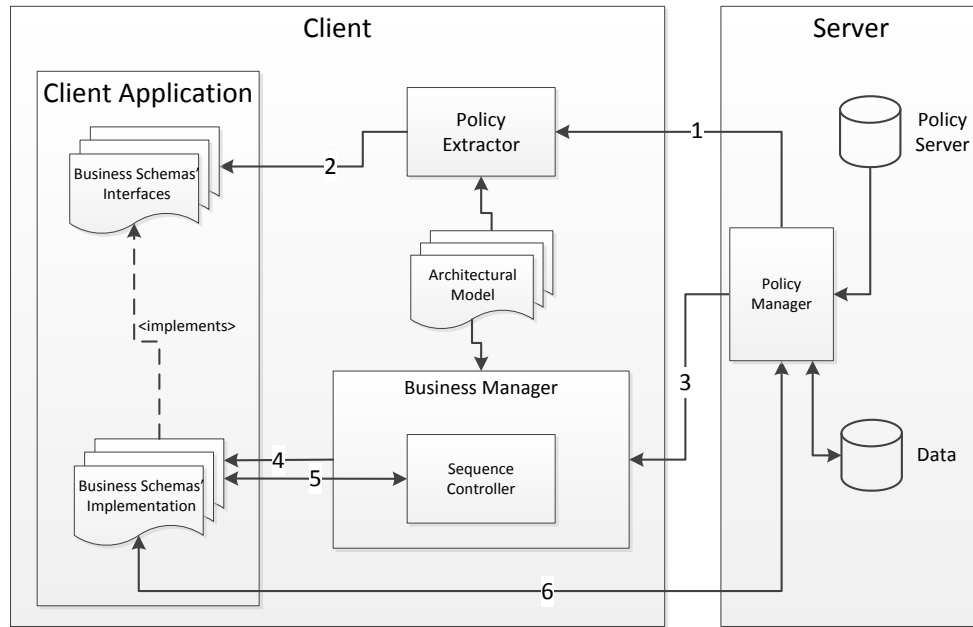


Figure 25. The S-DRACA architecture.

4.1.4 Static implementation of the RBAC policies

In this section we present our solution to statically implement the Business Schemas, used by a developer to access the information in the database. The extension can be formalized by several approaches and it depends entirely on the practical scenario at hand.

We decided to store the Business Schemas' interfaces in the policy server as a jar file (a jar file is an archive with the class files and some other information). When the Policy Extractor tool wants to generate a new set of Business Schemas interfaces, it starts by connecting to the Policy Manager, authenticates, and then sends a *GetBus* message. The *GetBus* message, shown in Figure 26, is intended to request the list of authorized Business Schemas and CRUD expressions, as well as the authorized sequences (with the revoke lists and list of allowed CRUD expressions, for each Business Schema in a sequence position).

First the *GetBus* message is sent, and the Policy Manager sends back a session ID that is randomly assigned to the client when it authenticates and is used to execute queries on the database. Then the number of Business Schemas is sent, so the client knows how many to read, and then the list

itself is sent. The list has the Business Schema URL and the list of valid CRUD expressions, which the size is also sent followed by each CRUD ID, session ID and the statement itself. The CRUD session ID is the temporary ID that the client can use to execute that CRUD statement and it is associated with the client session ID sent previously. This process of executing a query based on a temporary session ID is explained later in section 4.4.2. Then the status of the Sequence Controller is received, this means that the Sequence Controller can be deactivated if it is not required. Finally the information regarding the sequences are received, it starts with the number of sequences, followed by the list of sequences. Each sequence sent has an ID, the size of the sequence and each position has the list of revoked Business Schema and the list of allowed CRUDs.

Then the Policy Extractor sends a *GetJar* message, which requests the jar file with the Business Schema interfaces from the policy server. The Policy Manager retrieves the jar file and sends it back to the client, which reads each Business Schema interface using Java Reflection and generates the source code of the interfaces that the application is allowed to use.

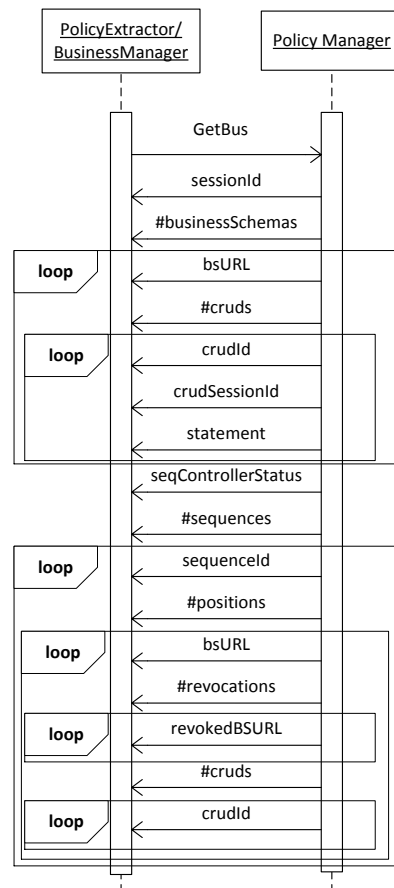


Figure 26. GetBus communication process.

The application can then use the Business Manager to request the instantiation of a Business Schema and request operations with it. The developer now does not need to master the defined policies or the database schema, because the Business Schemas can abstract the schema by providing

methods to access the authorized data only. The Business Manager, when is being initialized at runtime, sends the same *GetBus* message as the Policy Extractor, but this time this information is also used to validate the usage of a CRUD expression in a Business Schema and to validate if a Business Schema is instantiated or used out of order. The *GetJar* message is also sent and the retrieved Business Schema's interfaces are used to implement the interfaces. A generator is used for this process, which takes Business Schemas interfaces and generates the implementation of each method (i.e. *execute*, *moveNext*, etc.) for a specific database (several generators are supported, but only one can be used at the same time). For example, the Business Schemas receive a JDBC connection object in their constructor, which means that a Business Schema associated with a select statement generates a *ResultSet* when executed. The *moveNext()* method, shown in Figure 27, then is only required to call the *moveNext()* method on the *ResultSet*. The 'validationSourceCode' is the code responsible for checking with the Sequence Controller if the request can be executed or not.

These implementations are compiled and then loaded into the JVM when needed.

```

161 | str.append("public boolean moveNext() throws SQLException{\n");
162 | str.append(validationSourceCode);
163 | str.append("return rs.next();\n");
164 | str.append("}\n");

```

Figure 27. Sample code from the SQL Server generator.

4.1.5 Summary

In this section we presented an overview of DACA and S-DRACA and the S-DRACA architecture, which is based on the DACA architecture but with the modifications needed to support the enforcement of sequences of Business Schemas and to address some security vulnerabilities. Finally we discussed the mechanism used to implement the Business Schemas and the messages exchanged between the client and the server during this process.

4.2 Access control policies

In this section we will present the access control policies, used to generate the access control mechanisms shown in Figure 22. These policies are based on an extension made to RBAC that allows to define, for each role, the permissions a user has in the CRUD expression level. Furthermore, these CRUD expressions are associated with Business Schemas, i.e. entities whose interfaces are created during the compilation process and implemented at runtime, that guarantee that the client application can only follow the defined access control policies. To provide an increased level of control to security experts while defining the access control policies, the extension also allows the definition of sequences of Business Schemas. The sequences restrict the order in which the applications can execute the authorized CRUD expressions.

This section is divided as follows: Section 4.2.1 introduces our conceptual extension to RBAC, section 4.2.2 discusses the concretization of the conceptual model and section 4.2.3 defines the software architectural model that is used to enforce the extended RBAC policies.

4.2.1 Conceptual RBAC policy extension

The basic RBAC policy used to supervise requests to access data stored in RDBMS is comprised of: users, roles that can be hierarchized, permissions, delegations and actions. When a user wants to perform some action, he will only be able to do so if he plays the role that rules that action. In the end, the actions can be defined as one of the four main operations on database objects (tables and views): read, insert, update and delete operations.

In DACA, actions were formalized by what can be done on the direct and on the indirect access modes. This means that actions are the CRUD expressions that can be executed (direct mode) and also the operations that can be performed on the LDS (indirect mode). On one hand we have the granularity of the direct access mode, which is defined by each CRUD expression, on the other hand the granularity of the indirect access mode must be defined at the protocol level (read, insert, update and delete) and at the attribute level (except for the delete protocol, which must always be at the row level). The granularity at the LDS level provides full control for the security expert to define exactly which protocols are to be made available and, of those, which attributes should also be available. In terms of cardinality, each role is defined by a set of un-ordered CRUD expressions.

We kept this extension to the RBAC policy and further extended it to support the definition of sequences, which restricts the order CRUD expressions (through the Business Schemas) can be used. From this extension, security experts can now define new restrictions over the actions ruled by a role, particularly the ordered sequences of actions (execution of CRUD expressions) users can perform.

4.2.2 RBAC model extension

In this section will present a model that formalizes the presented extension to RBAC. In DACA, we started by analyzing the CRUD expressions, because every data access request starts through the direct access mode and only then can the indirect access mode be used. Furthermore, only a select (read) expression executed in the direct access mode allows the usage of the indirect access mode.

Each CRUD expression type (Select, Insert, Update, Delete) can be expressed by general schemas and each individual CRUD expression can be represented by specializing one of the general schemas. While the CLIs were being assessed, including JDBC, we found out that the schema of each CRUD expression type can be built upon a small set of smaller schemas. The functionalities provided by the smaller schemas are: only Select expressions return relations, all CRUD expression types can use runtime values for clause conditions, some CRUD expressions return the number of affected rows (Insert, Update, Delete) and finally, some CRUD expressions use runtime values for column values (Insert and Update). Some other perspectives for the LDS can also be elicited, such as some LDS are scrollable (where no restrictions are placed upon which the next selected row can be) while others are forward-only (where only the next row can be selected).

To address this set of smaller schemas, the general schema needs to be flexible and adaptable. This challenge was solved by the design of entities, previously introduced as Business Schemas. Business Schemas are responsible for hiding the actual direct and indirect access modes driven by the access control policies. After some research, the cardinality between the Business Schemas and the CRUD expressions was found to be many-to-many. This means that one Business Schema can manage several CRUD expressions and that one CRUD expression can be managed by several Business Schemas. To explain why let us consider two Select expressions:

1. SELECT column1 FROM table;
2. SELECT column1 FROM table WHERE column2 < 1024;

We can analyze first the direction “one Business Schema → many CRUD expressions”. From the direct access mode perspective, both expressions are exactly the same: both are select expressions that take no runtime values and the schemas of the returned relations are identical. If the security policy for both expressions is the same then the Business Schema can be reused. The other direction “one CRUD expressions → many Business Schemas” can be more easily explained. When different security policies are applied to the same CRUD expression, it has to be managed by several Business Schemas because a single Business Schema can only provide one security policy.

This previous extension allows us to define which Business Schemas and CRUD expressions are authorized for each user through their roles, but to be able to define the sequences, in which those Business Schemas and CRUD expressions can be used, and what their life-cycle is when a sequence moves forward one position we had to further extend the RBAC model.

This new extension must take into account two main aspects. The first aspect is the functionality to connect Business Schemas and, therefore, to build sequences of Business Schemas. The second

aspect is related to the life-cycle of Business Schemas when the sequence moves forward to the next Business Schema.

We start by presenting the first one of those aspects. To formalize our approach of sequences (S) we will use directed graphs (usually known as digraphs) of Business Schemas. In short, one directed edge (e) connects one source vertex to a destination vertex, where each vertex (v) is one Business Schema. From a general digraph, it is possible to define several different paths (i.e. a sequence of vertices that are connected in pairs by directed edges). This level of freedom can potentially be a cause for concern, especially where security is imperative. Hence, and although our model is defined using digraphs, some strong restrictions need to be established in order to prevent the usage of digraphs in its most general form. Notwithstanding this fact, we can start the design phase of the sequences using generic digraphs, although we will have to identify the paths that are compliant with the access control policies. A path is nothing more than a sequence of vertices which are connected by one edge only (except the last one, which has no edge). Only these valid paths can be used and assigned to roles. This means, then, that users authorized to play a role: 1) can execute the Business Schemas (vertices) defined by the associated path and no other and 2) in the order they are defined in the path (direct edges). The assignment of several paths to one role is permitted and our model does not enforce any restriction at that level. This can be important in situations where a role is defined to control the use of several forms, each one controlled with its own path, this way avoiding the need to define one role for each form.

Some definitions are now introduced. A root vertex (r) is the vertex where a path starts. A leaf vertex (l) is, by opposition, a vertex with no edges and, therefore, is the last vertex of a path. A front vertex (f) is the vertex where the sequence is currently running. We now present the properties that the paths in our model possess. The properties of paths are:

- one path comprises one and only one root vertex;
- one path comprises one and only one leaf vertex;
- self-edged vertices are not allowed in paths;
- loop-back vertices are not allowed in paths;
- any vertex of a digraph can be the root vertex of a path;
- any vertex of a digraph can be the leaf vertex of a path;
- any vertex of a digraph can appear zero, one or more times in one path;
- adjacent vertices in paths must also be adjacent vertices in the parent digraph and connected with the same direct edge.

Next we present a simple example of a digraph, shown in Figure 28. It comprises 5 vertices and 6 edges, where one of them is a self-edge (on vertex C). It is possible to define an infinite number of paths from this digraph, due to the loopback vertices (B to A) and the buckle (self-edge) on vertex C. Three of the possible paths that can be defined from Figure 28 are shown in Figure 29.

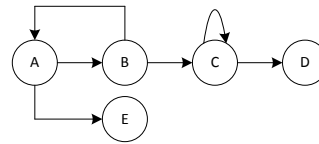


Figure 28. Example of a digraph.

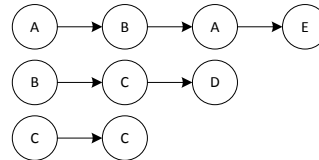


Figure 29. Examples of valid paths derived from Figure 28.

The second aspect is related to life-cycle of Business Schemas when a sequence moves to the next Business Schema. The basic idea is that when the sequence moves to the next Business Schema, it is up to the security manager to decide which Business Schemas are to remain active (i.e. their instances run normally) from the list of active Business Schemas and which are not to remain active (their instances are running but their methods now raise an exception). The process to disable Business Schemas instances is hereafter referred to as the revocation process. In our model, each Business Schema has an associated list with all the previous Business Schemas to be revoked and a subset of the allowed CRUD expressions for that Business Schemas, this way also allowing a security expert to disable specific CRUD expressions in certain positions of a path.

Finally, we present a possible and simplified model to formalize our proposal, shown in Figure 30. From it we can see that one role comprises one or more sequences (paths). The quaternary association indicates that a sequence comprises one or more Business Schemas (vertices) and that each sequence position (an entry in the sequence table) has associated a list of Business Schemas to be revoked and the subset of valid CRUD expressions to authorize. Finally, a Business Schema manages one or more CRUD expressions.

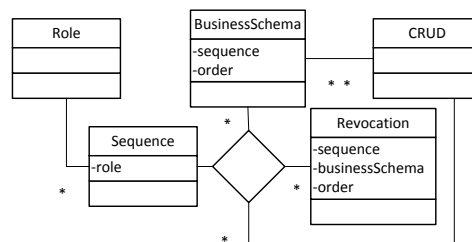


Figure 30. Extension for the RBAC model.

Figure 31 shows the implemented RBAC model. Parts of it comes from DACA, which includes: the subjects (Ses_Subjects), the hierarchy of roles (Rol_Roles), permissions (Per_Permissions), applications (App_Applications), delegations (Del_Delegations), sessions (Ses_Sessions), Business Schemas (Bus_BusinessSchema) and the CRUD expressions (Crd_Crud). To support the definition of

sequences of Business Schemas we added: sequences (Seq_Sequences), Business Schema's aliases (BSA_Alias) and revocations (Rev_Revocation).

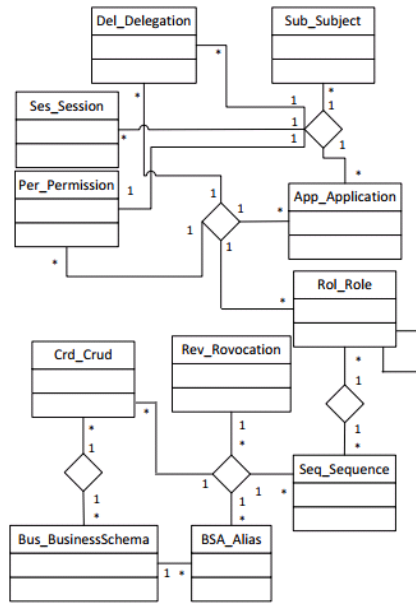


Figure 31. The implemented RBAC model.

It is clear that this implementation is derived from the extension presented in Figure 30, where we instead of associating Business Schemas to roles directly, we associate them with sequences that are associated with Business Schemas (one authorized per sequence position), revocation lists (the Business Schemas to revoke) and CRUD expressions (the subset of the valid CRUD expression for the Business Schema that is authorized). Another change that can be easily seen is the introduction of Business Schema aliases. They are meant to differentiate several instances of the same Business Schema in the sequences, so that when the application developer needs to get the next Business Schema in a sequence, only the valid options are available. If the same Business Schema were to be used, e.g. Business Schema A in Figure 28, it would have to have the option to obtain any of the possible Business Schemas that comes next, i.e. Business Schemas B and E in the example, which would require the developers to master the defined sequences. The problem and the solution are later addressed in section 4.3.

4.2.3 Software architectural model

In this section we present the software architectural model, shown in , for building the access control mechanisms from the extended RBAC model. The presented architectural model represents the implementation of one role. It is up to each system architect to decide how to expand it to support several roles. Moreover, it is focused on how to implement RBAC mechanisms and not how to build complete and feasible implementations. For example, the architectural model does not address key issues such as the scrolling policy on LDS and database transactions. These and other issues are out of the architectural model context. We start by describing the Business Schema interface, herein known

as `IBusinessSchema`, which is the most complex entity. From it we will present and describe the architectural model. This interface, as we can infer from what has been already presented, needs to cope with the two access modes. The functionalities to be provided depend mainly on the CRUD expressions type and on the necessary runtime values. This is translated into the architectural model where the `IBusinessSchema` extends two interfaces: direct access mode interface (IDAC) and the indirect access mode interface (IIAM).

4.2.3.1 Direct access mode interface

This interface manages the direct access mode. Depending on the type of CRUD expressions and on the runtime values, it can extend 1, 2 or 3 interfaces:

- `IExecute` - This interface is mandatory. It is responsible for the execution of CRUD expressions of any type and also for setting the runtime values for clause conditions.
- `ISet` - This interface is used with Insert and Update expressions when there is the need to set runtime values for columns.
- `IRows` - This interface is used only with Update, Insert and Delete expressions to notify applications about the number of affected rows.

4.2.3.2 Indirect access mode interface

This interface manages the indirect access mode. Depending on the mechanisms to be implemented, it can extend at most four interfaces:

- `IRead` - This interface is mandatory. It can comprise services to read any sub-set of attributes of returned relations.
- `IUpdate` - This interface is only available if the established access control policies authorize the attributes of LDS to be updated. In this case, only the updatable attributes can be updated.
- `IInsert` - This interface is only available if the established access control policies authorize the insertion of new rows on LDS. In this case, only the insertable attributes can be inserted.
- `IDelete` - This interface is only available if the established access control policies authorize the rows of LDS to be deleted.

Regarding the relation between Business Schemas and, Roles and CRUD expressions, we can see from that the architectural model is consistent with the RBAC model. Please remember that the architectural model represents the implementation of one role only. The model says that one role comprises one or more Business Schemas and each Business Schema comprises one or more CRUD expressions. From the presented architectural model and also from the RBAC model, security components can be automatically built, a process shown in Figure 33. To achieve this goal, a tool is

necessary to automate the process. It is not part of our proposal but the tool is a key component to transform the modeled RBAC policies into security components.

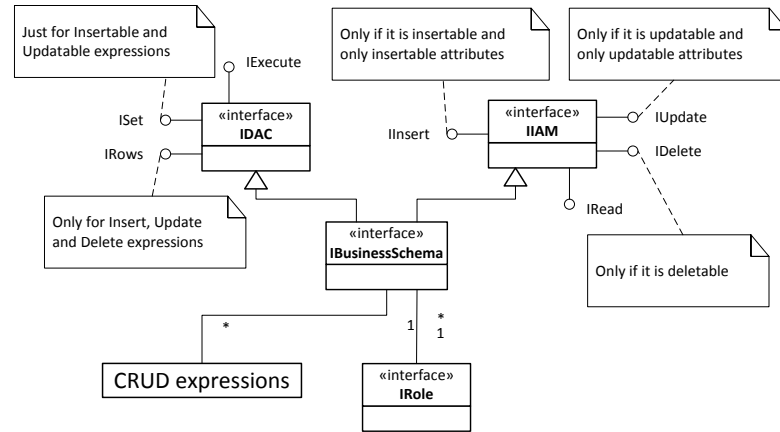


Figure 32. Software architectural model for one role.

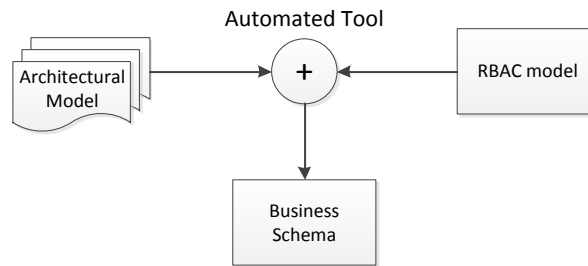


Figure 33. Automated building process of Business Schemas.

4.2.4 Summary

In this section we presented the extension proposed to the RBAC model in the access control policies layer and the software architectural model. When this software architectural model is used together with the defined policies in the extended RBAC model, it allows the generation of security components that enables client applications to access the sensitive data while relieving the programmers from having to master the database schema or the access control policies.

4.3 Sequence controller component

In this section we will present the sequence controller component. The idea of sequence control is to enable a security expert to define the order in which the Business Schemas can be executed and which CRUD expressions can be used in each step of each sequence. Figure 29 shows a simple graph encoding of three possible sequences. The sequence controller, given the information provided in that figure, would be able to enforce the application to instantiate the Business Schemas in that order. To provide further functionality, the sequence controller can receive a list of Business Schemas to be revoked in each step of each sequence, along with the list of CRUD expressions that can be used and it will also enforce that.

Figure 25 show that after the application receives the implementation of the Business Schemas in step 4, each action that the application requests will be validated with the sequence controller (step 5). If the Business Schema is requested to execute a CRUD expression that is not authorized for that step in the sequence being followed or if the Business Schema has been revoked, as explained in section 4.2.2, then the sequence controller will not authorize the Business Schema to execute that action and an exception will be generated. To control the execution of the Business Schemas and to obtain the reference to the next Business Schema in the sequence there are two possible solutions: using an orchestration based solution or a choreography based solution.

This section is divided as follows: Section 4.3.1 will present the conceptual model of the sequence controller, section 4.3.2 will present possible implementation solutions found, section 4.3.3 presents the details of the solution adopted and section 4.3.4 summarizes the contents of this section.

4.3.1 Conceptual model

In this section we will present the conceptual model for the sequence controller. Our approach is to define sequences of Business Schemas that can only be executed in that order (see Figure 34). For each sequence position we have the Business Schema that is to be authorized and the associated lists of Business Schemas for revocation and CRUD identifiers that are allowed to be used with the authorized Business Schema. The RBAC model extension presented in section 4.2.2 already accounts for this model as it can associate with a role multiple sequences. Each sequence is then associated with multiple aliases that reference the Business Schemas, which is explained further in section 4.3.2.4, and also with a revocation table for the revocation lists. The allowed CRUD expressions are associated with the Business Schemas and they provide a way to authorize only a subset of them at each position.

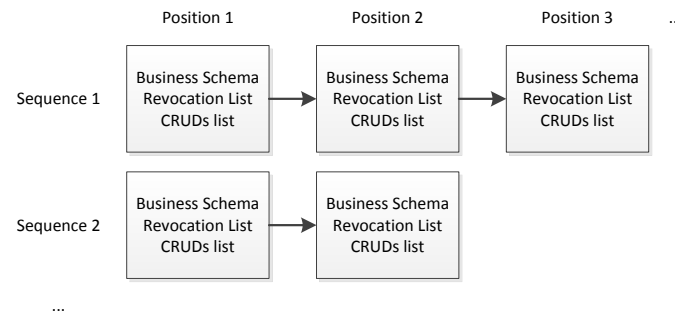


Figure 34. Sequence controller conceptual model.

4.3.2 Implementation solutions

In this section we will present the studied implementation solutions of the sequence controller component. To implement the sequence control functionality we needed to choose one the two possible approaches shown: orchestration or choreography. After some thought and experimentation we came to the conclusion that obtaining a reference to the next Business Schema from each Business Schema is a more intuitive way to use this feature.

We will now present the solutions that we analyzed that led to our final solution. The first solutions explored the orchestration approach by allowing the application to request the next Business Schema in the sequence to the class with the user's role security data structures. The last solution tried the choreography approach and the application would then retrieve the reference to the next Business Schema from the Business Schemas.

This section is divided as follows: Section 4.3.2.1 will discuss a solution based on enumerated classes declared in the role's security data structures, section 4.3.2.2 will discuss a similar solution but using references to the next Business Schema instead of enumerated classes, section 4.3.2.3 will present a distributed solution on the Business Schemas and section 4.3.2.4 will discuss the problems of each solution and which was chosen.

4.3.2.1 Using the role's security data structures interface with enumerated classes

In this solution, enumerated classes are used, i.e. classes with an ordinal name. This allows grouping the Business Schemas by their position in the sequences.

Figure 35 shows an example of this approach in practice. We have used three different Business Schemas: IS_Orders, which allows to select data from the Orders table, IS_Customers, which allows to select data from the Customers table and II_Orders, which allows to insert data into the Orders table.

We also defined two different sequences: Sequence 1 (S1), which is composed of only the IS_Orders Business Schema and Sequence 2 (S2), which contains the IS_Customers followed by the II_Orders Business Schemas.

The number of enumerated internal classes inside the role's security data structures interface depends on the sequence with the greatest number of Business Schemas, since there is an enumerated class for each possible position. This means that our example has two internal classes (lines 9 and 16) due to sequence S2 having two Business Schemas.

```

8 | public abstract interface Role_IRole_B1 {
9 |     public abstract class First {
10 |         public static final Class<IS_Orders> s_orders_S1 = IS_Orders.class;
11 |         public static final int s_orders_S_Orders_byShipCountry = 1;
12 |         public static final Class<IS_Customers> s_customers_S2 = IS_Customers.class;
13 |         public static final int s_customers_S_Customers_all = 2;
14 |     }
15 |
16 |     public abstract class Second {
17 |         public static final Class<II_Orders> i_orders_S2 = II_Orders.class;
18 |         public static final int i_orders_I_Orders_withCostumerID = 3;
19 |     }
20 | }

```

Figure 35. Security data structures using enumerated classes

Note that this approach bundles together all Business Schemas and CRUD expressions that are used in the same position in every sequence in the same internal class. The internal class *First* (line 9 to 14) contains both IS_Orders and IS_Customers Business Schemas. To differentiate the sequence that each one belongs to, the name of the sequence is added to the name of the Business Schema reference (lines 10, 12 and 17). To know which CRUD expressions can be used with which Business Schema, the name of the Business Schema is added to the CRUD expression's name (lines 11, 13 and 18). To avoid this, a data structure could be used that would contain the Business Schema reference and the allowed CRUD expressions.

Figure 36 shows an example of the usage of this approach. The application would use the Business Manager (variable *manager*, lines 71 and 75) to instantiate the Business Schema. To indicate which Business Schema to instantiate the role's security data structure interface would be used. The first Business Schema would be requested using the *First* internal class, indicating the Business Schema and the CRUD expression to use (lines 71 and 72). Then the instantiated Business Schema could be executed (line 73) and other operations requested. To instantiate the next Business Schema in the sequence, the same process would be required but this time the application uses the *Second* internal class (line 75 and 76). The Business Manager has the responsibility to authorize the instantiations and to validate the execution of every request made to each Business Schema, a process detailed in section 4.3.3.

```

71 | S_Cust = manager.instantiateBS(Role_IRole_B1.First.s_customers_S2,
72 |     Role_IRole_B1.First.s_customers_S_Customers_all, session);
73 | S_Cust.execute();
74 | // Read, Update, Insert, Delete ...
75 | I_Orders = manager.instantiateBS(Role_IRole_B1.Second.i_orders_S2,
76 |     Role_IRole_B1.Second.i_orders_I_Orders_withCostumerID, session);
77 | I_Orders.execute(/*values to insert*/);

```

Figure 36. Example of the usage of Business Schemas with sequence control using enumerated classes where customers are selected and a new order inserted.

This solution accomplishes the objective of controlling the execution and instantiation of the Business Schemas while allowing the application to follow the defined sequences. However, the idea that a sequence is being followed becomes lost, and the only reference to a sequence is in the names of the internal classes, i.e. *First*, *Second*, etc.). Moreover, the goal of each sequence is not clearly stated, although this could be solved by using a meaningful name to each sequence instead of *S1* or *S2*, like *insert_Order_for_Customer* in case of *S2*.

4.3.2.2 Using the role's security data structures interface with 'next' references

This solution uses the role's security data structures interface with 'next' references. The idea was to have a data structure that would contain the Business Schema reference, the allowed CRUDs to use and a reference to another class with the same structure that would create the sequences the application could follow, called "*next*". The interface's (see Figure 37) several classes would be named after the Business Schema they hold (lines 12, 17 and 23). The Business Schemas and sequences used are the same as the last solution presented in section 4.3.2.1. For the application to distinguish between the entry points for the sequences and the classes that reference intermediate Business Schemas, there would be attributes in the interface (lines 9 and 10), named *be_<X>*, that would reference the entry point for the sequence <X>. The classes are dynamically generated using the sequence's metadata received from the policy server database.

```

8 public abstract interface Role_IRole_B1 {
9     public final class_IS_Orders be1 = new class_IS_Orders();
10    public final class_IS_Customers be2 = new class_IS_Customers();
11
12    public class class_IS_Orders {
13        public static final Class<IS_Orders> s_orders = IS_Orders.class;
14        public static final int s_orders_S_Orders_byShipCountry = 1;
15    }
16
17    public class class_IS_Customers {
18        public final class_II_Orders next = new class_II_Orders();
19        public static final Class<IS_Customers> s_customers = IS_Customers.class;
20        public static final int s_customers_S_Customers_all = 2;
21    }
22
23    public class class_II_Orders {
24        public static final Class<II_Orders> i_orders = II_Orders.class;
25        public static final int i_orders_I_Orders_withCostumerID = 3;
26    }
27 }

```

Figure 37. Security data structures with next references.

Figure 38 shows how this approach would be used in practice. The application would have to access one of the internal classes using one of the attributes in the role's security data structure interface (line 74). Then the application would be able to use the Business Manager to instantiate the Business Schema refereciated by the internal class (line 75 and 76). The Business Manager would then be responsible for authorizing the instantiation and keep the sequence lifecycle information.

After the application receives the Business Schema instantiation, it can start requesting operations using it, like executing the underlying CRUD expression (line 77).

To request the next Business Schema in the sequence the application can access the *next* reference in the internal class obtained previously (line 79) and request the Business Manager to instantiate it (line 80 and 81). Finally the application can use the newly instantiated Business Schema to request operations (line 82).

```

74      b2 = Role_IRole_B1.be2;
75      S_Cust = manager.instantiateBS(b2.s_customers,
76      |      b2.s_customers_S_Customers_all, session);
77      S_Cust.execute();
78      // Read, Update, Insert, Delete ...
79      b2_2 = b2.next;
80      I_Orders = manager.instantiateBS(b2_2.i_orders,
81      |      b2_2.i_orders_I_Orders_withCostumerID, session);
82      I_Orders.execute(/*values to insert*/);

```

Figure 38. Example of the usage of Business Schemas with sequence control using next references where customers are selected and a new order inserted.

This solution accomplishes the objective of controlling the execution and instantiation of the Business Schemas while allowing the application to follow the defined sequences. However, we are required to use several data structures in the application in this solution: the role's security data structures interface, the internal classes declared in the role's security data structures interface and the Business Manager. Using this amount the classes just to use Business Schemas and request the next one in the sequence can become a little confusing. Another problem with this approach is that, since each internal class in the role's security data structures interface can only contain one "*next*" reference, two sequences that start with the same Business Schemas will require different internal classes for each one of them, which increases the number of internal classes considerably. In this case the name of the sequence is only in the name of the entry point reference (*b1* and *b2*), which does not clearly state the end goal of each sequence. Again, this problem could be solved by adding a meaningful name to them, but after using the entry point reference it is no longer used and can be hard for someone reading the source code to understand what is being done if the sequence is long enough.

4.3.2.3 Using the Business Schemas

Using the Business Schemas themselves we were able to apply the idea of choreography, which allows a simpler usage and understanding by the developers, at the cost of implementing the sequence control logic over all the Business Schemas instead of a single interface. The benefit of this is that the developers no longer have to use an additional class to use S-DRACA, relying directly on the preexisting Business Schemas. It does, however, make the generation of the Business Schemas slightly more complex, since the sequence control logic also has to be generated at compile time in the interfaces.

```

7 public abstract interface Role_IRole_B1 {
8     public static final java.lang.Class<II_Orders>
9         i_orders = II_Orders.class;
10    public static final int
11        i_orders_I_Orders_withCostumerID = 2;
12    public static final java.lang.Class<IS_Orders>
13        s_orders = IS_Orders.class;
14    public static final int
15        s_orders_S_Orders_byShipCountry = 1;
16    public static final java.lang.Class<IS_Customers>
17        s_customers = IS_Customers.class;
18    public static final int
19        s_customers_S_Customers_all = 3;
20 }

```

Figure 39. Security data structure comprising information about *Role_B1*.

In the role's security data structures interface we now have the Business Schemas references and CRUD expressions declared in an unsorted manner (see Figure 39). We will no longer use this interface directly to instantiate Business Schemas, so it does not require any sort of human readable structure. Instead we wrap the instantiation of each entry point of each sequence in a new class named Factory, as shown in Figure 40. After instantiation, the Business Schema is used in the same manner as before.

```

60 S_Cust = factory.get_S_Customers_all(session);
61 S_Cust.execute();

```

Figure 40. Interaction between application and the Factory class.

Figure 41 shows how the Factory class instantiates a Business Schema. To remove the need for the developer to match each Business Schema to an allowed CRUD expression like the in the previous proposed solutions, we created a service in the Factory for each allowed Business Schema – CRUD expression pair, using the name of the service to distinguish each of them, e.g. line 25 shows a service called *get_S_Customers_all* which implies that the Business Schema instantiated will be *S_Customers* with the CRUD expression that returns all tuples in the table. Like before, it isn't clear what the end goal of the sequence will be, so adding a meaningful name to this service and the services in each Business Schema to obtain the next one in the sequence would solve this problem. This Factory class is generated by S-DRACA as well from the access policies retrieved at compile-time.

```

25 public IS_Customers get_S_Customers_all(ISession session)
26     throws LocalTools.BTC_Exception {
27     return businessManager.instantiateBS(
28         Role_IRole_B1.s_customers,
29         Role_IRole_B1.s_customers_S_Customers_all,
30         session);
31 }

```

Figure 41. The Factory requests an instance for a Business Schema.

Obtaining the next Business Schema in the sequence is a manner of calling the respective service from the current Business Schema being used. If the Business Schema *S_Customers*' instance was obtained previously (from the Factory class or another Business Schema) then the process of requesting the next Business Schema is shown in Figure 42, using the *get_S_Orders_by_shipCountry*

(line 65), which returns an instance of the S_Orders Business Schema, associated with the CRUD expression that selects the orders of a customer using the ship country as a filter (line 66).

```

65 | S_Orders = S_Cust.get_S_Orders_by_shipCountry(session);
66 | S_Orders.execute(S_Cust, "Portugal");

```

Figure 42. The S_Orders Business Schema provides an edge method to step forward to next Business Schema.

Figure 43 shows how the instantiation of the next Business Schema is done in each intermediate Business Schema. It first validates the execution of the method in the current Business Schema (lines 20 to 24), which is done in every public service of the Business Service. It uses the sequence identifier and the URL of the current Business Schema. If the Business Manager does not validate the execution an exception is thrown (line 22).

After that the Business Schema requests the next one from the Business Manager (line 26). Since the name of the service already defines the Business Schema to be instantiated and the underlying CRUD expression, the developer does not need to indicate them and they are passed to the Business Manager in the generated service (line 27 and 28). The active sequence is an identifier for the context of the sequence being executed and it is explained later in section 4.3.3.

Both Figure 40 and Figure 42 show how to use the Business Schemas with the sequence controller mechanism deployed.

```

17 | @Override
18 | public IS_Orders get_S_Orders_by_shipCountry(ISession session)
19 |     throws BTC_Exception {
20 |     if(!businessManager.validateExecution(
21 |         activeSequence, "S_Customers.S_Customers")) {
22 |         throw new IllegalStateException(
23 |             "S_Customers was used out of order!");
24 |     }
25 |
26 |     return businessManager.instantiateBS(
27 |         Role_IRole_B1.s_orders,
28 |         Role_IRole_B1.s_orders_S_Orders_byShipCountry,
29 |         session,
30 |         activeSequence
31 |     );
32 | }

```

Figure 43. Validation process for the instantiation process of Business Schemas.

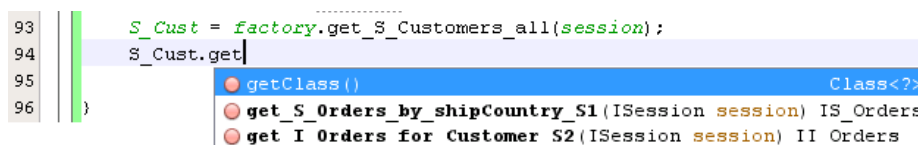
This approach is easier to use because we no longer have to manually indicate which CRUD expression we want for each schema by using the semantically more rich set of services. A similar solution for the previous approaches would require a similar Factory class, but since the sequence control logic isn't distributed it would have to offer all the services for getting every single allowed Business Schema – CRUD expression pair. Nor only that, but the developer would have to be able to distinguish the entry points to each sequence and the services for intermediate Business Schemas. The services names could be used for that, but having all the services in one place could prove confusing with complex policies.

4.3.2.4 Common problems and the chosen solution

Having in mind the problems that each solution has, we tried to choose the approach that: 1) does not involve too many different data structures, 2) gives the idea that a sequence is being followed, 3) the next Business Schema required is easily identifiable and 4) prevents attempts to instantiate Business Schemas in the wrong order.

The first two solutions are not the best considering the above objectives, either they have many data structures or they do not show the idea that a sequence is being followed. The third objective can be easily reached in any approach by using meaningful names in the services/variables provided to instantiate the Business Schemas.

The last objective is not reached by any of the previous approaches and does not have a trivial solution. The first two solutions based on orchestration can't reach this objective since it is the developer that requests the Business Manager directly for the Business Schemas' instances and he can request the wrong Business Schema or CRUD expression by mistake. In the third approach, where the methods exist in the Business Schemas themselves, Business Schemas can be used in more than one sequence. This implies that a Business Schema will have a service for each next Business Schema and allowed CRUD expression pair in every sequence it appears unless it is the leaf node (i.e. there is no next Business Schema). We can see now that a Business Schema can have services to request the next Business Schema for multiple sequences, which means that a developer can mistakenly call the wrong one. This is where the Business Schemas' aliases, shown in section 4.2.2, come into place. By having different versions of the same Business Schema under different aliases it is possible to use the aliases in the sequence policies instead of the Business Schemas. This allows the client application to generate slightly different versions of the same Business Schema, which only allows the developer to request the next authorized Business Schema alias in the sequence being executed.



```

93  S_Cust = factory.get_S_Customers_all(session);
94  S_Cust.get
95
96  }

```

The autocomplete dropdown menu shows the following options:

- `getClass()` `Class<?>`
- `get_S_Orders_by_shipCountry_S1(ISession session)` `IS_Orders`
- `get_I_Orders_for_Customer_S2(ISession session)` `II_Orders`

Figure 44. Using the Business Schema's to get the next one without aliases.

Figure 44 shows an example of the third approach, i.e. using the Business Schemas to request the next ones in the sequence, when no aliases are used. The example has three Business Schemas: IS_Customers, IS_Orders and II_Orders. The Business Schemas are used in two different sequences: sequence 1, defined by IS_Customers followed by IS_Orders, and sequence 2, defined by IS_Customers followed by II_Orders. When we want to request a Business Schema (line 93) the Business Schema has to be bound to a sequence, which should be declared in the method name or the documentation, but from the autocomplete feature of the IDE we still have access to the services to get the next Business Schemas from both sequences.

```

95 | S_Cust_S1 = factory.get_S_Customers_all_S1(session);
96 | S_Cust_S1.get
97 | S_Cust_S1.get
98 | }
    | get_S_Orders_by_shipCountry_S1(ISession session) IS_Orders
    | getClass() Class<?>

```

Figure 45. Using aliases in place of Business Schemas for sequence 1.

The aliases solution allows to prevent that problem by returning a different Business Schema specialization, i.e. one of its aliases. Figure 45 show, in practice, how this solution works. By selecting at the Factory the sequence we require (line 95) we are presented with an alias of the Business Schema, which is functionally identical but only allows to request the next valid Business Schema alias (line 96). Figure 46 shows the same outcome for the other sequence when using aliases. Henceforth, when we mention the Business Schemas, we also reference their aliases. The original Business Schemas alone will be referenced as the Master Business Schemas.

```

96 | S_Cust_S2 = factory.get_S_Customers_all_S2(session);
97 | S_Cust_S2.get
98 | S_Cust_S2.get
99 | }
    | get_I_Orders_for_Customer_S2(ISession session) II_Orders
    | getClass() Class<?>

```

Figure 46. Using aliases in place of Business Schemas for sequence 2.

4.3.3 Implementation of the sequence controller

In this section we will present the implementation details of the sequence controller component. Considering the solutions explored for controlling the sequence of execution of the Business Schemas and the problems associated with each of them, we decided to implement the third solution where we use the Business Schemas to provide the service to request the next one in the sequence. Because of this we require an entity, i.e. the sequence controller, which can authorize the execution of requests and instantiation of Business Schemas to avoid replicating the sequence access policies in each Business Schema instance. It is important to note that the sequence controller does not execute the requests made to each Business Schema, but instead is responsible for providing each Business Schema with the decision whether they can execute a request and if a Business Schema can be instantiated given a sequence context which is henceforth known as an *ActiveSequence*.

Figure 47 shows the implementation details of the sequence controller interface and the required data structures required to support the functionality at the Business Manager that implements the interface. Two data structures are defined: the *SequenceEntry* data structure and the *ActiveSequence* data structure. The *SequenceEntry* holds a Business Schema URL, the associated list of Business Schemas URLs to revoke and the list of allowed CRUD identifiers. The *ActiveSequence* data structure holds the list of active Business Schemas URLs, the associated sequence identifier being executed and the relative position in that sequence. Furthermore, the Business Manager possesses the list of all sequences, which maps a sequence identifier with a list of *SequenceEntry* objects, and the list of active sequences, which maps the *ActiveSequence*'s identifiers to each *ActiveSequence* object. An identifier of the *ActiveSequence* is given to each Business Schema instead of the actual object to prevent possible misuses. As for the services provided by the sequence controller, it contains services

for configuration, lifecycle manipulation and execution decision. The configuration services provided are: `addBSToSequence()`, `removeSequence()`, `setControlStatus()` and `clearControl()`. The services for lifecycle manipulation are `authorize()` and `requestNewActiveSequence()`. As for the execution decision service there's the `validateExecution()`.

To configure the sequence controller we have the service `addBSToSequence` which adds a new `SequenceEntry` to a sequence and requires the sequence identifier, the Business Schema alias URL, a list with Business Schema aliases' URLs as the revoke list and the list of allowed CRUD identifiers. The service `removeSequence` requires a sequence identifier and removes the associated sequence from the controller. The service `setControlStatus` requires a boolean value and allows the sequence controller to be enabled or disabled. When disabled every request is allowed. The service `clearControl` resets the sequence controller to a state where no sequences are configured.

To manipulate the lifecycle of a sequence we have the `requestNewActiveSequence` service, which initializes a new `ActiveSequence` and returns the associated identifier. We also have the `authorize` service, which is used to authorize a new instantiation of a Business Schema and if it is authorized it steps forward the sequence position. A Business Schema is only authorized if it is the next Business Schema in the associated sequence. It also revokes the Business Schemas that are alive that have their URL in the associated revoke list.

Finally, to validate the execution of Business Schemas we have the `validateExecution` service. It requires an `ActiveSequence` identifier and the Business Schema URL trying to execute a request. Only if the Business Schema is authorized, i.e. it has been authorized for instantiation and not yet revoked, will the service validate the execution and return *true*.

This section is divided as follows: Section 4.3.3.1 will discuss the initialization phase of the sequence controller and section 4.3.3.2 will discuss the management of the lifecycle of a sequence.

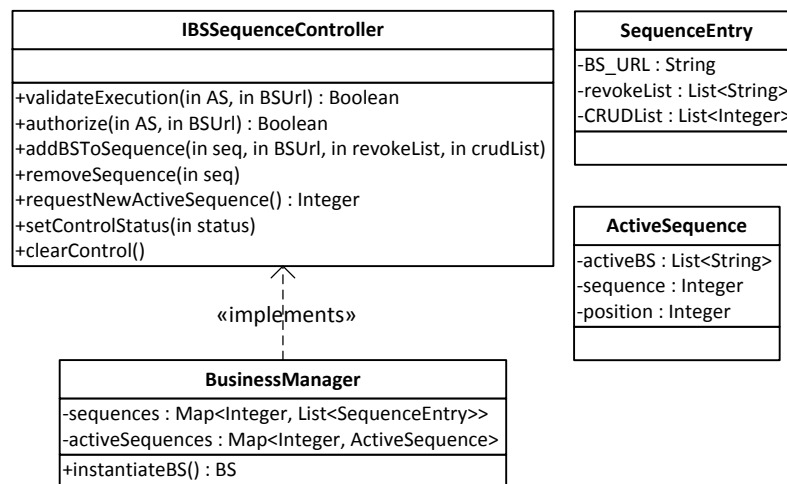


Figure 47. Implementation of the sequence controller component in the Business Manager.

4.3.3.1 Initialization

As described in the section 4.1.4, when the application requests the policies from the Policy Manager using the “getBus” method, the Policy Manager replies with a list of each Business Schema that the application has permission to use, along with their CRUD expressions.

The process has been changed so that the client also receives the information required for the sequence controller, i.e. the status of the sequence controller (enabled or disabled), the sequence’s identifiers, the Business Schema’s URL for each sequence position and its associated revoke and allowed CRUD identifiers lists.

Since the client has to instantiate the Business Manager to use S-DRACA, it requests this information to implement the Business Schemas and to configure the sequence controller with the received data. After receiving the sequence controller status from the Policy Manager, the setControlStatus service is called to set it to the enabled or disabled mode. After that, when a full entry for a sequence position is received from the Policy Manager, i.e. the Business Schema URL of the Business Schema to authorize, the list of Business Schemas to revoke and the list of allowed CRUD expressions allowed, the service addBSToSequence is called to configure the sequence controller with the new sequence entry.

4.3.3.2 Sequence control lifecycle

After configuring the sequence controller, the Business Schemas can start being instantiated and their execution validated. Figure 48 shows the software model for the sequence control mechanism.

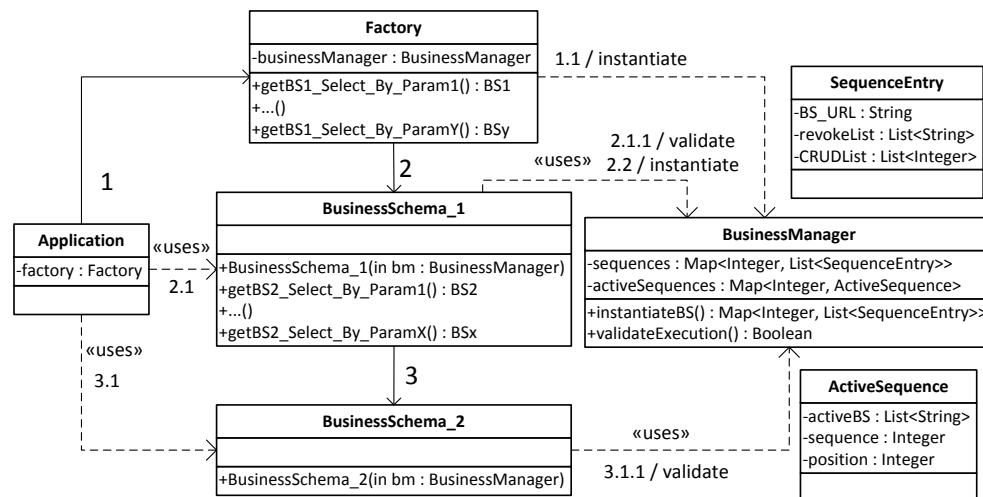


Figure 48. Software model for the sequence control mechanism.

The application starts by requesting a Business Schema to the Factory class (1). The Factory class then forwards the request of the Business Schema instantiation to the Business Manager (1.1), which authorizes the instantiation and then the first Business Schema becomes available to the application (2). The application then uses the Business Schema by requesting operations on it (2.1),

which are validated with the Business Manager (2.1.1). The application, when it requires it, can request the next Business Schema to the one it instantiated previously, which forwards the request to the Business Manager that authorizes the instantiation (2.2). The next Business Schema then becomes available to the application (3), which can use it to request operations on (3.1). These operations are also validated with the Business Manager (3.1.1). The process repeats for the length of the entire sequence.

Figure 49 shows a detailed sequence diagram of the instantiation process and the operations the application can request to the instantiated Business Schema.

As previously explained, the application starts by requesting a Business Schema from the Factory class, which forwards the request to the Business Manager. The Business Manager then requests a new ActiveSequence identifier and tries to authorize the Business Schema. If the authorization is granted, then the Business Schema is instantiated. A process that checks if the constructor can be executed using the same code that appears in Figure 43 (lines 20 to 24). The instance of the Business Schema is then returned to the application.

Each operation that the application requests to the Business Schema is validated in the same way and, if valid, it's executed. The result is then returned to the application.

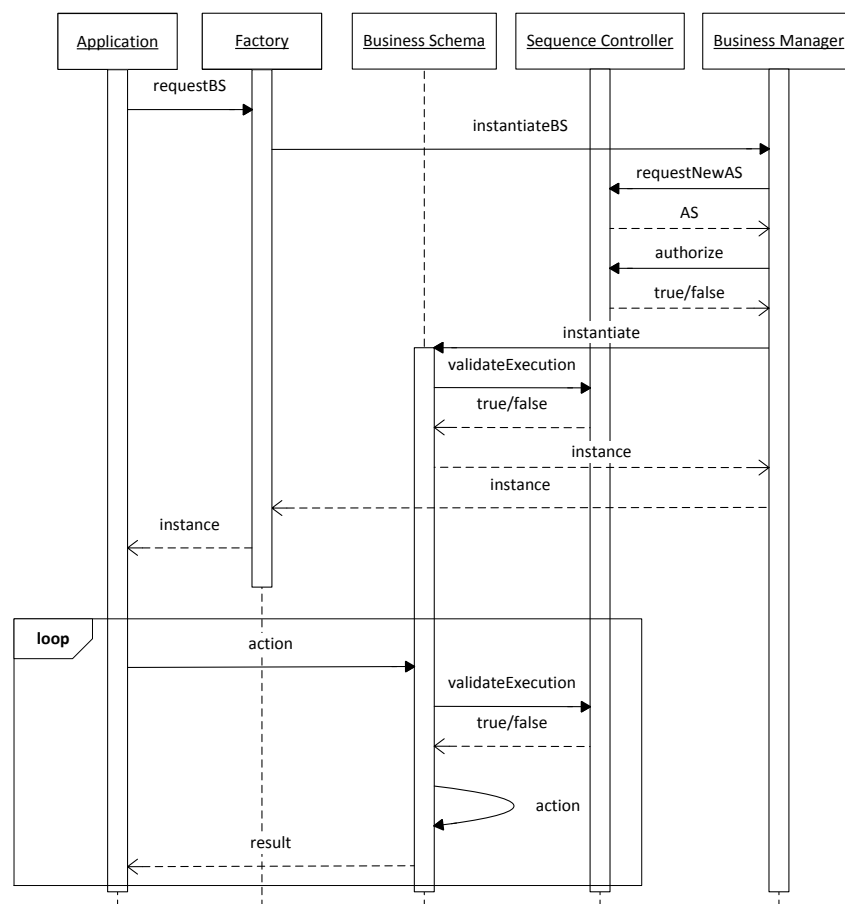


Figure 49. Sequence diagram of the sequence control mechanism for requesting the first Business Schema and making requests.

Figure 50 shows another detailed sequence diagram of the process of requesting the next Business Schema from an existing Business Schema. The application requests the next Business Schema to the one it already possesses. The execution is validated and, if valid, the instantiation request is made to the Business Manager. If it authorizes the instantiation the Business Schema is instantiated and the instance returned to the application. The process in the Business Manager is almost the same, except that a new ActiveSequence identifier is not requested this time, since the instantiation must refer to an existing ActiveSequence, which is passed as a parameter on the Business Schema instantiation request.

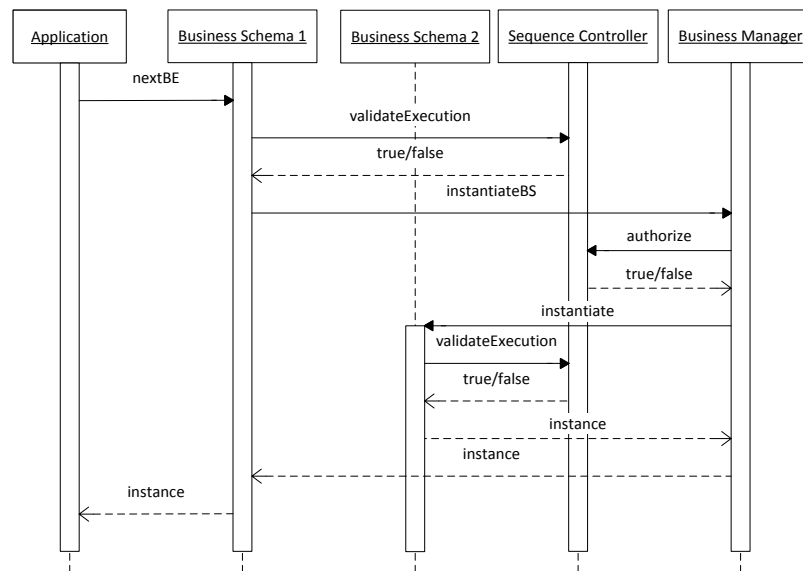


Figure 50. Sequence diagram of the sequence control mechanism for requesting the next Business Schema.

To exemplify the process, Table 8 shows a possible set of data on roles, Business Schemas and associated CRUDs that are allowed to be executed. Table 9 shows two possible sequences that could be encoded in the proposed solution for Role_B1. Each position in each sequence then has the authorized Business Schema, the revocation list and the CRUDs authorized to be executed.

An application using the role B1 would be able to access two different sequences. The sequence with the identifier 2 has a length of 3. The application would request the Business Schema I_Orders to the Factory class which would associate the CRUD expression 3. After the application has finished using it the application can then request the next Business Schema, i.e. S_Customers. We can see that by instantiating this Business Schema the revoke list, when applied, will revoke the first Business Schema. If the application tries to execute any service from the first Business Schema an exception will be raised.

Finally, the application can request the final Business Schema S_Orders to the S_Customers. Note that originally S_Orders has two authorized CRUD expressions, but the sequence data only contains one of them in the authorized CRUD list. Because of this S_Customers only provides a service to get the S_Orders associated with the CRUD expression 1.

Table 8. Roles, Business Schemas and CRUD expressions.

Role	Parent Role	BS	CRUD		
			Id	Ref	Expression
Role_A					
Role_B1	Role_A	S_Orders	1	byShipCountry	Select * From Orders Where CustomerId = ? and ShipCountry = ?
			2	byFreightLimit	Select * From Orders Where CustomerId = ? and Freight < ?
		I_Orders	3	withCustomerID	Insert Into Orders Values (?,?,?,?,?,?,?,?,?,?)
		S_Customers	4	all	Select * From Customers
Role: Role Reference BS: Business Schema alias Id: CRUD Identification Ref: CRUD reference Expression: CRUD Expression					

Table 9. Roles, sequences, revocation and CRUD lists.

Role	Sequence	Position	Sequence Entry		
			BS	RL	CRUDs
Role_B1	1	1	S_Customers		4
		2	S_Orders		1, 2
	2	1	I_Orders		3
		2	S_Customers	I_Orders	4
		3	S_Orders		1
Role: Role Reference Sequence: The sequence identification Position: The position in the sequence BS: Business Schema alias RL: The revocation list CRUD: The list of authorized CRUDs					

4.3.4 Summary

In this section we discussed the conceptual model of the sequence controller that would enable a security expert to define sequences of Business Schemas for the roles. The users then would be required to follow those sequences. To provide a finer control over the sequences, the security expert can define which CRUD expressions are available for a given Business Schema in a sequence and which Business Schemas should not be usable anymore after the client application reaches a certain point in a sequence. Then we discussed different solutions for the implementation and tried to find one that enforced the defined sequences while being easy to use by programmers. We chose to use a solution based on service choreography, where a programmer can get the next Business Schema in a sequence from the one that he already possesses. We finally discussed how this solution was implemented and integrated with S-DRACA.

4.4 Security layer

In this section we will present the proposed security layer. The proposed extension to the RBAC model provides developers with the ability to easily write client applications that use information stored in a data store, without the need to master its schema or the RBAC policies defined. We haven't, however, concerned with the security aspects of the architecture as a whole.

DACA had a very naïve client authentication mechanism, no data encryption and the client had access to the CRUD expressions being used. The latter exposes some of the underlying schema of the database, effectively leaking information that can be regarded as sensitive.

The authentication mechanism was based on PAP, a very rudimentary protocol which is nothing more than a message from the client to the server, which in our case is the Policy Manager, with the client's username and password in the clear, i.e. anyone watching the network could read this information, and a response from the server indicating if the authentication was successful or not. We propose several authentication methods that require different levels of trust in the network, the client and the server.

The data encryption from/to the database is only an issue when the network cannot be trusted. In these cases the client and the server, during authentication, establish a secure channel (with data encryption and at least client authentication). We propose using this channel for the client to communicate with the database.

Finally, to prevent the client from accessing the CRUD expressions used on the database we propose a mechanism protect the queries where the client uses a stored procedure to execute the desired query using only an identifier. This stored procedure, called RemoteCall, does not provide any information about the underlying database schema nor about the query executed.

Another security concern is the ability for the client to execute unwanted queries on the database. DACA was vulnerable to Java Reflection, which can expose the connection objects used and, through them, an attacker can easily execute unwanted queries. One way to prevent this in S-DRACA is to make use of stored procedures, instead of CRUD expressions for the defined queries for each Business Schema, and configuring the database to allow only the execution of stored procedures. This solution would prevent the execution of unwanted queries, but the whole sequence control mechanism could still be bypassed and, if the stored procedures do not receive a session identifier, even unauthorized ones for the application's role could be executed. Another solution could make use of the fact that Java Reflection cannot be used through sockets, so having the generated implementation code in a proxy on the server side and have the generated code on the client side request the operations to the proxy would solve this problem, but that would quite possibly introduce performance issues and bottleneck concerns. This security problem was not addressed in S-DRACA. The implementation details of standard security mechanisms can be found in Appendix A. Implementations that required deviations from the standard are discussed in this section.

This section is divided as follows: Section 4.4.1 will discuss the authentication methods implemented, section 4.4.2 will present a method of protecting the CRUD expressions from being accessed by intruders, section 4.4.3 will present a method for accessing a database securely while the client application only has access to credentials with no permissions associated and section 4.4.4 will summarize the presented contents in this section.

4.4.1 Authentication and data encryption

In this section we will present the authentication and data encryption mechanisms used or developed for S-DRACA. Security in communications starts generally with an initial authentication handshake, and sometimes even integrity protection and/or encryption of data. There isn't an authentication method that is the best of every scenario. Different protocols have different trade-offs and some security threats are more probable in some scenarios than in others (Kaufman et al., 2002b). For this reason we provide several authentication methods that can be used depending on the scenario at hand.

The authentication mechanism in S-DRACA for the client side followed a modular approach, where an abstract authentication class, shown in Figure 51, defines the basic authentication operations and delegates the authentication protocol to the classes that extend it.

<i>DacaClientAuthenticator</i>
#listeningPort : Integer
#socket : Socket
+authenticate(in appName : String, in username : String, in password : String) : Socket
#getHostIP() : String
#getHostListeningPort() : Integer
#getSalt(in username : String, in socket : Socket) : String
#hashPasswordBytes() : byte[]
#hashPasswordToString() : String
#secondStepAuthentication() : Boolean

Figure 51. The abstract *DacaClientAuthenticator* class.

The only abstract method is the *authenticate* method. Extending classes must implement this method, in which they can use the socket to communicate with the server and implement a specific authentication method. Other methods, implemented in the abstract class *DacaClientAuthenticator*, can provide a way to hash user's password to a byte array or a string with the byte array values in hexadecimal and to request the user's salt that was used to hash the password. Furthermore, it also implements the second step authentication, which will be explained later in this section.

The server has each authentication mechanism implemented in a class that handles the client's connections.

Note that in the beginning of each authentication mechanism, the client knows the application name (appName), its IP address and its listening port, used by the server to connect to the client to disseminate the changes made to the defined policies. The server also has access to each client's hashed password and the salt used.

When the server authenticates the client then the process of generating the random session identifier and the random CRUD expression identifiers mentioned in the previous section takes place.

This section is divided as follows: Section 4.4.1.1 presents the hash-based password authentication protocol, section 4.4.1.2 presents the challenge-response authentication mechanism, section 4.4.1.3 presents the data encryption methods and section 4.4.1.4 discusses the integration of a TFA with the CC. Keep in mind that only the non-standard implementation details are presented in this section and that the implementation details of standard security mechanisms can be found in Appendix A.

4.4.1.1 Hash-based Password Authentication Protocol

The hash-based password authentication protocol is based on the password authentication protocol where the username and password of the client is sent in clear text over the network. Instead of the password being sent in clear text, a hash of the password concatenated with a salt is sent. This way, even if an intruder eavesdrops on the communication, he will only be able to get the original password using a dictionary attack.

This method, however, does not prevent the intruder from authenticating with the server, since the intruder knows the username and the hashed password, but it protects the user of the client application if he uses the same password for other services.

This method of authentication provides very little protection against an intruder that eavesdrop the communication, since it isn't encrypted and the hashed password can be used to impersonate the client, so its usage should be restricted to scenarios where the network itself is trusted and secure. It is, however, the simplest authentication method implemented and provides the lowest communication overhead of all.

4.4.1.2 Challenge-Response based protocol

The challenge-response authentication mechanism is one where the client does not send its password (hashed or not) through the network. Instead, the server sends the client a challenge and waits for a response. This authentication mechanism also allows mutual authentication, since the client can also send a challenge of its own and validate the response. This mechanism, if implemented correctly to prevent reflection attacks (the attack, which is explained in section 3.3.4.5, not to confuse with the reflection functionality of programming languages like Java), can prevent an intruder from impersonating a client or the server until he can discover the user's password using a dictionary attack, both preventable by using a strong password.

This authentication method is more secure than the hash-based password authentication protocol. Not only it provides mutual authentication, but it also prevents an eavesdropper from collecting the user's password (at least not without a dictionary attack) and thus preventing the impersonation of the clients or the server.

It does suffer from a bigger communication overhead, and when the hash-based password authentication protocol only required the server to compare values, this solution requires the creation of challenges and the calculation of responses, as well as comparing the responses to check if they match. It is, however, a good solution when the network can't be trusted but the data transmitted to/from the database can't be sensitive, because it still travels through the network in clear text.

4.4.1.3 SSL/TLS

To solve the issue of the data being communicated in clear text we turned to SSL/TLS. The normal use of SSL/TLS relies on certificates to allow the endpoints to agree on a symmetric key for the session. Another way to use SSL/TLS is to use the anonymous Diffie-Hellman method, which does not authenticate either the client or the server but avoids the usage of certificates. We propose a solution using a server side certificate, a variation of the anonymous Diffie-Hellman method with added mutual authentication, and two other variations, SPEKE and SRP, that we could not implement but we still analyze.

This section is divided as follows: Section 4.4.1.3.1 discusses using SSL/TLS with certificates, section 4.4.1.3.2 discusses a variation of the Diffie-Hellman key exchange protocol that uses a pre-shared key (PSK) to authenticate the client and the server and sections 4.4.1.3.3 and 4.4.1.3.4 introduce similar standard solutions that were not possible to implement due to restrictions on the programming language level.

4.4.1.3.1 Server Certificate

The authentication mechanism using SSL/TLS with a server certificate didn't require a lot of changes to work with S-DRACA. A private key had to be loaded in the server side and a public key certificate distributed to the client. For this mechanism to remain secure, the client must take great care in not allowing an intruder to replace it, or the server could be impersonated. Note that the client remains unauthorized with this method alone because it does not use a private key that the server can validate with a public key certificate. To prevent this and to authenticate the client, we use the challenge-response mechanism after the encrypted channel is created to authenticate both the client and the server.

This method has an even higher communication overhead, because the challenge-response mechanism is used in this method along with the request for upgrading the communication channel and the whole SSL/TLS handshake process. Unlike the previous mechanisms, however, it encrypts the data sent and received, protecting it from eavesdroppers in the network and authenticates both the client and the server. MITM attacks are also prevented by having the public key certificate distributed beforehand.

4.4.1.3.2 Pre-Shared Key + Anonymous Diffie-Hellman

The usage of certificates places extra responsibility on the clients, whom have to maintain the public key certificate of the server secure. This fact may require a level of trust in the clients that is not compatible with the business policies. Even with the added challenge-response mechanism, replacing the public key certificate means that an intruder can still impersonate the server and obtain the user's username, which can be used on the real server to obtain a challenge-response pair. Then it's a matter of running a dictionary attack to find the password that transforms the challenge into the response.

The anonymous Diffie-Hellman method uses the Diffie-Hellman key exchange protocol so that the two endpoints can agree on a shared secret that is then used to derive the session key for encrypting the rest of the communication. This is, however, vulnerable to MITM attacks since the endpoints are not authenticated before the key exchange takes place. Doing it after, such as using the challenge-response mechanism, is ineffective since the intruder only has to relay the messages back and forth.

We analysed a small variation introduced in the Diffie-Hellman protocol that aims to change the agreed key the same way on both endpoints using a PSK that only the real client and server should know. This PSK that we used was the hash of the password of the client and, if the agreed key was K , then the new agreed key used to calculate the session key becomes $K' = \text{hash}(K + \text{hash}(s + \text{salt}))$, being s the client's password and salt the salt value used to calculate the hash of the password.

This transformation of the agreed key cannot be performed by an intruder in a MITM attack, since he should not know the password of the client. This way, when no MITM attack is in effect, both the client and the server can communicate using the SSL/TLS channel as normal, but if an MITM attack is in effect, then the intruder cannot decrypt the data coming from the channel. He cannot relay the data back and forth either, since the agreed key a client gets is different from the one the server gets, so the server also cannot decrypt the data.

This method, which we called PSK-SSL, still contains a big vulnerability, however. In a MITM attack scenario, after the agreed key has been hashed and the new session key calculated, the client will send a message to the server with a known structure, such as a *GetBus* message. The intruder can, then, perform a dictionary attack to transform the known agreed key with the client and attempt to decrypt the message until the known structure is obtained. However, it is possible to detect that a MITM attack has occurred. If a client does not receive a response from the server inside a time window then it's best to assume a MITM attack has occurred and a warning stating that the user's password should be changed should be raised and the current password disabled.

We faced a problem while trying to implement this method. Since the SSL/TLS implementation in Java is closed and can't be extended we couldn't directly change the agreed key K . We relied on the reflection functionality of Java to access the Java implementation of the SSL/TLS protocol and to perform the necessary changes to make it work.

Figure 52 shows how an anonymous SSL/TLS connection is established using Diffie-Hellman and where the agreed key is altered. First, an SSL socket is created (line 32) and the desired cipher suit

is specified (line 33). The cipher suit is a named combination of authentication, encryption and MAC algorithms and a pseudorandom function. The cipher suit *TLS_DH_anon_WITH_AES_128_CBC_SHA* states that the key exchange protocol to be used is Diffie-Hellman (DH) and no authentication will be made (anon), which removes the need to use certificates since they are used to provide authentication.

```

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 52. Server socket creation process with the password-authenticated key exchange.

The server then waits for a client to connect (line 34). When a client connects, the server saves the reference to the *handshaker*, which is an internal object of the *clientSocket* that handles the handshaking process of the SSL/TLS protocol (line 35). *ReflectionUtils* is a class that provides several functionalities based on reflection, where *getFieldValue(obj, fieldName)* retrieves variable with the name *fieldName* from the object *obj*. It is required to save this reference because after the handshaking process finishes, its reference is set to null.

Then a normal handshaking process takes place (line 36), after which the agreed key is changed (line 37).

```

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);

```

Figure 53. Client socket creation with the password-authenticated key exchange.

Figure 53 shows the same process from the client point of view. First we attempt to connect to the server (line 46). Since the cipher suit used is disabled by default we have to enable it in the client as well (line 47). Then we save the reference to the *handshaker* object for the same reason we did in the server (line 48) and then start the normal handshake process (line 49). Finally we change the agreed key using the same process as the one used by the server (line 50).

This handshake process uses Diffie-Hellman to agree on a *preMasterKey* (the shared secret) which is then used to derive a *masterKey* from which the read and write ciphers are initialized. Since the *preMasterKey* is disposed of after the *masterKey* is created we couldn't change it, so we decided on changing the *masterKey* instead since it remains available.

Only changing the *masterKey* isn't enough, however, because the read and write ciphers that read and write into the communication channel have already been initialized. We will now explain the complete process required to change the *masterKey* and update the read and write ciphers. 1) First, we obtain the current private state of the socket (should be *cs_DATA* since the handshake has finished). 2) Obtain the *masterKey* from the SSL session. 3) Hash the *masterKey* bytes with the shared secret (the

user's hashed password) and truncate the output to the original size of the masterKey. 4) Invoke the *calculateConnectionKeys* private method from the *handshaker* object that calculates the connections keys for the read and write ciphers. 5) Set the *handshaker* reference in the socket, since it has been set to null after the handshake. This is required for step 7. 6) Set the socket state to *cs_HANDSHAKE*. The methods invoked in step 7 assert that the socket is still handshaking. 7) Call the private methods *changeReadCiphers* and *changeWriteCiphers* declared in the socket class. 8) Cleanup by setting the socket's state back to the original value and the socket's handshaker reference back to null.

The previous process requires the usage of Java Reflection in every step except step 3, because the variables set and methods invoked in the socket are private.

Our reliability on the reflection functionality has the big problem of destroying the abstraction created by the public interface. Any changes made to the Java SSL/TLS internal structure could potentially break the solution since it is dependent on the implementation.

4.4.1.3.3 Simple Password Exponential Key Exchange

This method is not currently supported by the SSL/TLS implementation of Java and we could not implement it because it required the Diffie-Hellman protocol itself to be modified. Furthermore, Java does not allow us to extend it with a new key exchange protocol.

This variation prevents MITM attacks like the previous variation, but has the big advantage of also preventing the usage of dictionary attacks. This is true because the generator, which is the value based on the password, is set to the power of the secret random number of the client/server and it's these values that are sent over the network (see Table 4, values *A* and *B* are related), which means that an attacker who is able to read and modify all messages between Alice and Bob cannot learn the agreed secret (value *s*) and cannot make more than one guess for the password in each interaction with a party that knows it. An intruder, to test a password, needs to calculate the public value to send (*A* or *B*), which depends on the password, and send it to the other party. If the password is wrong the generator *g* will be different and the agreed secret *s* will also be different. When the involved parties were to confirm that they both have arrived at the same key (by performing a different hash of the agreed key on both ends) the communication would be taken down.

It remains as future work to provide this authentication mechanism due to its advantages over the previous authentication methods.

4.4.1.3.4 Secure Remote Password

We did not implement the SRP authentication method, because it is not natively supported by the Java SSL/TLS implementation and no other providers for Java were found that did have SRP integrated in SSL/TLS, although Bouncy Castle("The Legion of the Bouncy Castle," n.d.) is in the process of implementing it. Another reason is that it requires a different set of user information in the database, like a password verifier, to be able to authenticate it, so the presence of the hashed password, required for the other authentication methods, would nullify the benefit of using a password verifier.

Although it isn't currently implemented, it is our goal to do so in the future due to the major benefits that it has when used in non-trusted networks with no mutual trusted third party certificate authority.

4.4.1.4 Two Factor Authentication

A TFA is an authentication method that requires two elements for authentication, i.e. a user to access his bank account at an ATM must use his bankcard (something that he **has**) and then input a PIN (something that he **knows**). In our case we decided to allow the usage of the CC, a smartcard that contains certificates and cryptographic capabilities for authentication and signing. These certificates can be read without the user having to input the required PIN but to perform any authentication operation, such as signing data, it has to be requested to unlock the card.

The authentication certificate, among other data, contains a unique identifier (BI) which is stored in the database along with the user's hashed password and salt value when the user is registered. This identifier is then later used to match a received public key certificate to the user that is trying to authenticate.

4.4.2 Query protection

In this section we will present our solution to the problem of the users being able to disclose the SQL statements used and learning about the database schema, by pushing them to the server side instead of residing in the clients, using the RemoteCall stored procedure and the session identifiers that we mentioned in the beginning of section 4.4. The RemoteCall stored procedure is defined in the database and allows each client to execute any CRUD expression that is defined in the policy server through an identifier. This effectively pushes the CRUD expressions to the server side, protecting them from malicious users that could target DACA since it would send the CRUD expressions to the client side. The stored procedure can be parameterized, which allows the execution of parameterized CRUD expressions. To avoid SQL injection attacks (Halfond, 2006), i.e. the unrestricted execution of SQL statements by embedding them into authorized ones, the execution of the defined CRUD expressions in the RemoteCall stored procedure uses, in the case of SQL Server, a special command called *sp_executesql*, which takes a parameterized CRUD expression and a list of parameters. Not only it is resilient against SQL injection attacks by treating the parameters differently from the CRUD expression, but it enables the RDBMS to optimize the queries.

The usage of the RemoteCall stored procedure involves two different phases: a preparation phase and an execution phase. The preparation phase is where randomly generated identifiers are associated with each CRUD expression that a client application can use and an equally random session identifier that is given to a client application when it first authenticates with the server. When the Business Schemas are instantiated, they automatically set the CRUD identifier and the session identifier in the statement that contains the RemoteCall stored procedure. The execution phase is where the client application executes the RemoteCall stored procedure, passing the respective CRUD

identifier that it wants to execute and the required parameters, if any. Since the result of the RemoteCall stored procedure is exactly the same as if the CRUD expression had been executed, no further changes were required in S-DRACA.

The *sp_executesql* command takes one (when the query isn't parameterized) or three arguments (when the query is parameterized). Its syntax is:

```
sp_executesql [ @statement = ] statement
[
    { , [ @params = ] N'@parameter_name data_type [ OUT | OUTPUT ] [ ,...n ]' }
    { , [ @param1 = ] 'value1' [ ,...n ] }
]
```

We can see that it always receives a statement, which is the query to execute. The other two arguments are optional, of which the first contains some metadata about the parameters of the statement to execute, i.e. their names and data types in a single string. The name is in the format @ParameterName and the data type is the name of a SQLServer name type, such as *int* or *nvarchar*. The third argument is also a single string with the parameters' values in the format @ParameterName = <parameter_value>. Several parameters are separated using a comma in both arguments. A query executed using the *sp_executesql* that retrieves every order from an Orders table where the Customer identifier is greater than a certain value can be written as: "EXEC sp_executesql N'SELECT * FROM Orders WHERE CustomerID > @CustomerID', N'@CustomerID nchar(5)', N'@CustomerID = 10';"

Figure 54 shows the signature of the RemoteCall stored procedure. It receives three arguments: a SessionID, which is the random identifier assigned to the client application, the QuerySRID, which is the random query's session identifier and the Params, which by default is an empty string and contains the parameter values for the CRUD expression being executed.

```

9 ALTER PROCEDURE [_remote].[RemoteCall]
10     @SessionID INT,
11     @QuerySRID INT,
12     @Params NVARCHAR(MAX) = ''
13 AS
14 BEGIN
```

Figure 54. Signature of the RemoteCall stored procedure.

The RemoteCall stored procedure requires three tables: Queries, SessionQueries and Operands. The Queries table stores the actual CRUD expressions. The SessionQueries table maps a generated session identifier and a generated random CRUD identifier to the real identifier of each CRUD expression. The Operands table stores the list of operands required by each CRUD expression and their data type, information that is required by the *sp_executesql* command.

The RemoteCall stored procedure, upon being executed, performs the following operations: 1) retrieves the identifier of the original CRUD expression using the SessionID and the QuerySRID from the SessionQueries table, 2) retrieves the CRUD expression to execute from the Queries table, 3) builds the parameter definition string, which contains the name of each parameter and its type and 4)

executes the *sp_executesql* command with the CRUD expression retrieved, the parameter definition that was built and the @Params parameter that contains the actual values of the parameters.

Consider the next two CRUD expressions in the Queries table and the operands in the Operands table:

Table 10. Example of two CRUD expressions in the Queries table.

QueryRID	CRUD Reference	CRUD Expression
1	S_Customers_all	SELECT * FROM Customers
2	S_Orders_byShipCountry	SELECT * FROM Orders WHERE CustomerId = @CustomerId and ShipCountry = @ShipCountry

Table 11. Example of information in the Operands table for the Queries in Table 10.

QueryRID	Position	Name	Type
2	1	@CustomerId	nchar(5)
2	2	@ShipCountry	nvarchar(15)

A client application with permission to use both CRUD expressions, when a new session is started, is assigned the session identifier 12345678 and the SessionQueries table is added the following information:

Table 12. Example of the generated identifiers for a client session.

SessionID	QuerySRID	QueryRID
12345678	13572468	1
12345678	24681357	2

The queries that the client receives that he is able to execute are:

1. EXEC PolicyServer2_remote.RemoteCall @SessionID = ?, @QuerySRID = ?;
2. EXEC PolicyServer2_remote.RemoteCall @SessionID = ?, @QuerySRID = ?, @Params = '@CustomerId = ?, @ShipCountry = ?';

As previously stated, each Business Schema, when instantiated, automatically sets the value of the SessionID and the QuerySRID parameters. The first RemoteCall call does not have a @Params parameter, since the original CRUD expression is not parameterized, but the second one does because the original CRUD expression has two parameters.

If the second query were to be executed, the RemoteCall stored procedure would: 1) retrieve the identifier of the original CRUD expression from the SessionID = 12345678 and QuerySRID = 24681357 - @QueryRID = 2, 2) retrieve the CRUD expression - @Query = N'SELECT * FROM Orders WHERE CustomerId = @CustomerId and ShipCountry = @ShipCountry', 3) build the parameter definition string - @ParamsDef = N'@CustomerId nchar(5), @ShipCountry nvarchar(15)' and 4) execute the *sp_executesql* command - EXEC('EXEC sp_executesql N''' + @Query + ''' + @ParamsDef + @Params).

4.4.3 Secure data store communication

The Business Schema's implementation, shown in Figure 25, communicates with the Policy Manager to access the database. This was not the case in DACA, where the communication was made directly between the Business Schemas and the database. This was a big security vulnerability, since the user had access to the credentials used to access the database with the sensitive data, which allowed him to bypass the security components completely and execute any CRUD expression uncontrollably. Not only that, but the sensitive information was transmitted through the network in clear text, so any eavesdropper could see the information.

To prevent this problem we present in this section a mechanism where the user could connect to the database without ever knowing the real credentials used to access it, while preventing eavesdroppers from collecting any of the sensitive data. The idea is to use the previously created and authenticated communication channel to transmit the interaction between the client and the database with the sensitive data. Since we want to encrypt the communication, the channel has to use one of the SSL/TLS enabled authentication mechanisms presented.

Obviously the channel does not connect directly to the database, but to the Policy Manager, which would connect to the database and then relay the communication between it and the client.

We ran into some problems trying to implement this, however. The JDBC implementation does not allow us to use an existing socket to connect to the database. We used the DriverManager provided by Java, which receives a connection string. The connection string has the following syntax for SQL Server:

`"jdbc:sqlserver://[serverName[\\instanceName]][:portNumber]][;property=value[:property=value]]"`.

The *jdbc* part indicates the protocol to use, which in this case is JDBC. The *sqlserver* part allows the DriverManager to determine which driver should be used, which in this case is the JDBC driver provided by Microsoft. Then it receives the server name and the port number to connect to a target DBMS. Using the connection string and successfully connecting to the database is the only option available to get the Connection object, from which we can create statements to execute queries. The Connection object then contains the socket that was created when the connection to the database was made, which we can manipulate only through reflection since it is inaccessible from the public interface.

In short, we require a connection object so that the Business Schemas can execute the CRUD expressions and to get the connection object we have to establish a connection to the database, since we can't use an existing socket to do it, and then access the internals of the connection object to change the socket so that the existing authenticated and encrypted channel to the server is used instead.

To create the connection object we noticed that the real credentials aren't required, just the credentials to an account that has no permissions to execute queries will provide the client with the necessary connection object. Then, using reflection, the internal socket and input/output streams are set to those of the socket used to communicate with the server. The usage of reflection has the downside of making this process work only for the Microsoft SQLServer driver. Other drivers may use

other internal structure for the connection object, requiring different approaches to change the socket used.

The client then sends a *Data* message, changing the server's thread that is handling the client requests to data mode. In this mode the server connects to the database using the real credentials and, using reflection, relays the data that comes from the client to the database and vice versa.

It is possible to configure the DBMS to use certificates and establish connections using SSL/TLS, but that would require the creation of a second secure communication channel and the client would still have to use the credentials for the account with the permissions to execute CRUD statements.

The *Connection* class implementation for SQL Server has several variables, but one is of particular interest for our goal: a variable named *tdsChannel*. This is the variable that holds the socket, named *channelSocket*, two identical input streams, named *inputStream* and *tcpInputStream*, and finally two identical output streams, named *outputStream* and *tcpOutputStream*. When we say that they are identical, we mean that they hold the same reference. The input/output streams are the streams used by the socket to read and write data. With this information it is easy to understand how the communication to the database can be transmitted through the socket that is connected to the server (i.e. the Policy Manager) instead of the database itself.

First, after the authentication process and other communications are finished, a connection object is obtained from the client, using the dummy account with no permissions. After that, the client sends the string *Data* to the server, indicating that he wants to change to data mode. The client then takes the socket that he is using to communicate with the server and its input/output streams and sets them in the *Connection* object, which was obtained in the first step. Meanwhile, the server creates its own *Connection* object using an account with the required permissions and starts copying the data from the client's socket into the socket of the *Connection* object that he just created and the opposite communication direction in a separate thread. With this setup, the client can continue to use the original *Connection* object that he created to communicate with the database without ever knowing the real account's credentials.

4.4.4 Summary

This section presented the developed security layer that was integrated into our solution. This security layer was presented in three parts: the authentication and data encryption mechanisms, the mechanisms that protect the CRUD expressions from being accessed by intruders using Java reflection or by capturing network packets and a method to allow client applications to access data in the database knowing only some credentials that provides no permissions to access or modify the data.

4.5 Performance

In this section we will present the performance tests done to measure the performance of S-DRACA when applied in practice. One aspect that is important to know, when using an architecture that provides extended functionalities, is how much overhead that architecture introduces when used. This aspect is important when human interaction is present, like in the case of S-DRACA, where its usage by developers should not make the resulting application too slow that frustrates users. Generally a response time below 2 seconds is acceptable, as shown empirically in (Shneiderman, 1984).

DACA lacked such an analysis, and with the introduction of a more complex authentication mechanism and the client-database communication relaying through the Policy Manager in S-DRACA, we needed to understand the impact that those functionalities, together with the rest of the architecture, would take on the communication with the database.

To do this we propose to analyze three execution scenarios: 1) a single select statement (SELECT), 2) two select statements (TWO SELECTS) and 3) a select followed by an insert, another select, an update and another select statement (SISUS). Note that to simulate a real scenario as close as possible we read some of the data from each row returned by a select statement and some of the parameters in the insert and update statements have random values. Also, after each SISUS scenario execution, the inserted values are deleted to prevent slowness due to the table size.

We also tested these scenarios using various iterations counts, from a single iteration up to 10000. To be able to compare the results we also tested the direct approach, using JDBC without S-DRACA, and using the same prepared statement (SPS) and a different prepared statement (DPS) in each iteration, making sure that they are implemented as similarly as possible.

We also compared the initialization times of S-DRACA, since it has to authenticate, request the RBAC policies for the role of the user, generate the source code from it and compile it.

This section is divided as follows: Section 4.5.1 defines the test environment, section 4.5.2 shows the results of our tests during the initialization phase, section 4.5.3 shows the results of the tests during our execution phase, section 4.5.4 draws conclusions on the data obtained and section 4.5.5 summarizes the content presented.

4.5.1 Environment

First we will specify the test environment and the machine used to run the performance tests, shown in Table 13. Note that all unneeded programs and services were not running or disabled. Network connectivity was also disabled.

Table 13. Testing machine specification.

OS	Microsoft Windows 7 SP1
Architecture	x86_64
Motherboard	Pegatron Corporation G60JX (Socket 989)
CPU	Intel Core i7 720QM – 1.6GHz
Memory	8.00Gb Dual-Channel DDR3 – 666MHz (9-9-9-24)
Hard Drive	596Gb Toshiba MK6465GSX ATA Device (SATA)
RDBMS	Microsoft SQL Server 2012
Other Programs	Netbeans IDE

4.5.2 Initialization tests

In this section we will explain the initialization tests performed and show their results. To obtain the times we used the *System.nanoTime()* service, which uses the current JVM's high-resolution time source and returns its value with nanosecond precision, but not necessarily with nanosecond resolution. It is not related to the current time and it's only usable to calculate elapsed time. The tests compared the time it takes to obtain a simple JDBC connection object against the time it takes to fully initialize S-DRACA, to the point where we were able to begin instantiating Business Schemas.

First, we present the initialization times in Table 14, Table 15 and Table 16. The first column represents the time spent to connect to the database. The next three columns are linked to the initialization times for S-DRACA. They are, respectively, the configuration time, the instantiation time of the Business Manager and the connection time to the Policy Manager. The last column represents the S-DRACA total time. Figure 55 shows a visual representation of the total times for JDBC and S-DRACA in the tables.

Table 14. Initialization times for JDBC and S-DRACA when both relaying and TFA were used.

JDBC	S-DRACA			
Connection	Configuration	Instantiation	Connection	TOTAL
42	6881	0.0026	50	6931
26	6991	0.0019	20	7011
36	6453	0.0019	60	6513
43	6558	0.0019	25	6582
36	6696	0.0026	38	6734
61	6792	0.0026	26	6818
42	5772	0.0026	26	5798
37	4374	0.0026	25	4398
57	4372	0.0026	30	4402
42	6099	0.0024	33	6132

Table 15. Initialization times for JDBC and S-DRACA when neither relaying nor TFA were used.

JDBC	S-DRACA			
Connection	Configuration	Instantiation	Connection	TOTAL
18	3140	0.0013	11	3151
11	3036	0.0019	12	3048
9	3487	0.0019	8	3495
10	2565	0.0019	9	2574
10	2909	0.0019	7	2916
13	2733	0.0019	9	2742
16	2414	0.0019	8	2422
9	2337	0.0032	11	2348
13	3405	0.0019	47	3452
12	2892	0.0020	14	2905

Table 16. Initialization times for JDBC and S-DRACA when relaying was used but not TFA.

JDBC	S-DRACA			
Connection	Configuration	Instantiation	Connection	TOTAL
16	2583	0.0013	13	2596
17	3089	0.0013	17	3106
20	2694	0.0013	16	2710
12	2716	0.0019	13	2728
14	2464	0.0013	12	2477
15	2584	0.0019	10	2595
12	2528	0.0019	13	2541
17	2454	0.0013	11	2465
13	2566	0.0019	15	2581
15	2631	0.0016	13	2644

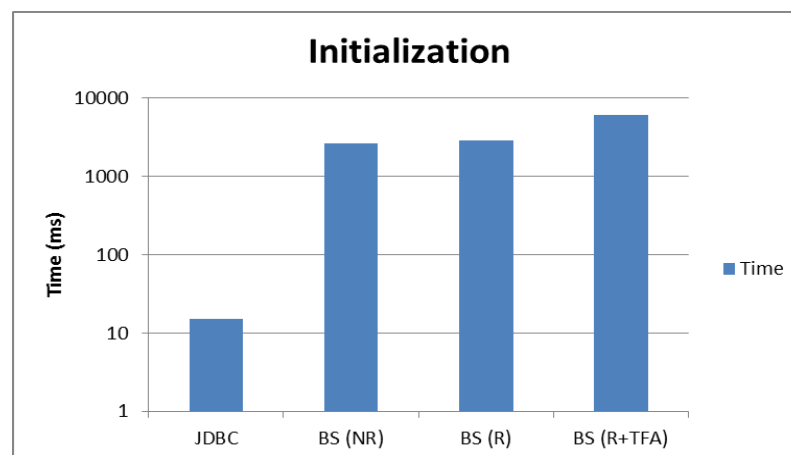


Figure 55. Graph of the initialization times in Table 14, Table 15 and Table 16.

4.5.3 Execution tests

In this section we discuss our performance tests and show the results obtained. We created several performance tests in order to test the performance in several dimensions: the number of test iterations, several scenarios and different cases for executing the tests, as explained earlier. The measurements were done with calls to the *System.nanoTime()* service as before.

For a given iteration we have three measurement values for each case - scenario combination and their average (in the light-grey rows). We use the averages to draw our conclusions on the architecture's performance in order to mitigate odd results that might appear. Figure 56, Figure 57 and Figure 58 also use these average values to build the graphs. Note that the usage of TFA does not impact these results, since the TFA mechanism does not introduce any overhead after it's completed, which is during the initialization, so it does not appear as a test case.

The first scenario (SELECT) is meant to verify the delays when a single select statement is used. The second scenario (TWO SELECTS) is meant to understand the delays involved when two different queries are alternated. The final scenario (SISUS) is meant to extract the delays involved with a more complex group of queries. The queries use random parameters where possible, to prevent possible optimizations by the RBDMS.

We start by testing a single iteration of each scenario, to get the delay of an atomic execution, and then we increment the number of iterations by factors of ten to simulate increases in load. The results of the performance tests are shown in Table 17.

Table 17. Performance execution times.

ITR	SELECT				TWO SELECTS				SISUS			
	BS (NR)	BS (R)	DPS	SPS	BS (NR)	BS (R)	DPS	SPS	BS (NR)	BS (R)	DPS	SPS
1	7	34	7	7	9	13	3	3	41	36	26	30
1	6	7	4	8	9	13	3	3	34	39	21	21
1	6	10	5	6	10	20	3	3	36	58	42	32
1	6	17	5	7	9	15	3	3	37	44	29	27
10	17	28	12	20	36	52	18	22	442	428	613	491
10	18	25	13	17	32	59	20	20	499	500	714	680
10	14	22	18	20	39	56	18	13	473	660	785	336
10	16	25	14	19	36	56	19	19	471	529	704	502
100	146	178	104	74	240	335	142	144	2079	2713	3218	2060
100	139	169	108	98	285	323	142	119	2178	2595	1952	2154
100	108	163	110	91	247	347	165	136	2457	2460	2167	3130
100	131	170	108	88	257	335	150	133	2238	2589	2446	2448
1000	942	1391	1090	729	2105	2977	1674	1053	17004	23062	21176	16600
1000	923	1561	1003	818	2007	2960	1595	1113	17686	22766	17699	18629
1000	915	1555	1023	673	2035	2973	1624	1897	20323	21602	20603	22476
1000	927	1502	1039	740	2049	2970	1631	1354	18338	22477	19826	19235

ITR	SELECT				TWO SELECTS				SISUS			
	BS (NR)	BS (R)	DPS	SPS	BS (NR)	BS (R)	DPS	SPS	BS (NR)	BS (R)	DPS	SPS
10000	9006	13455	9758	6362	18094	26366	15703	10711	181835	204776	192194	179348
10000	8697	13118	9724	6666	18266	26669	16203	11463	180734	204352	196014	177731
10000	8732	13374	9371	6616	18223	26688	16491	10716	192674	201777	207025	181943
10000	8811	13315	9617	6548	18194	26574	16132	10963	185081	203635	198411	179674

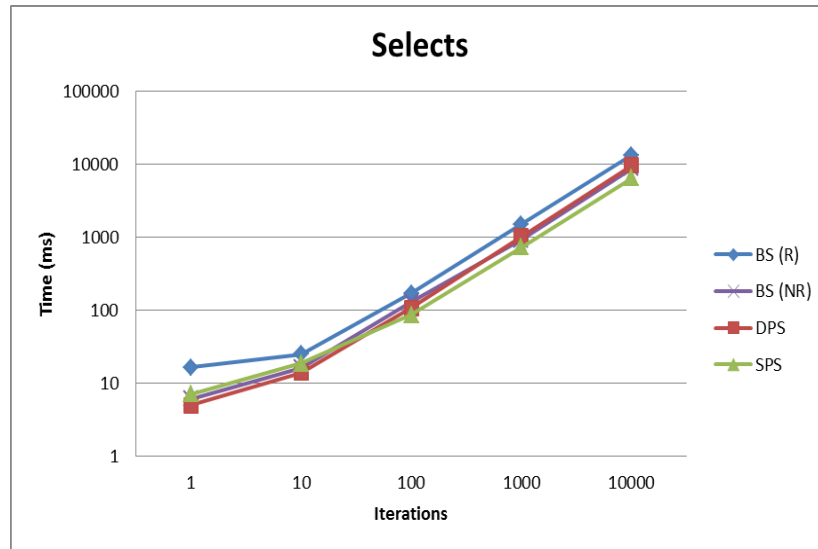


Figure 56. Graph of the execution times in Table 17 for the single select scenario.

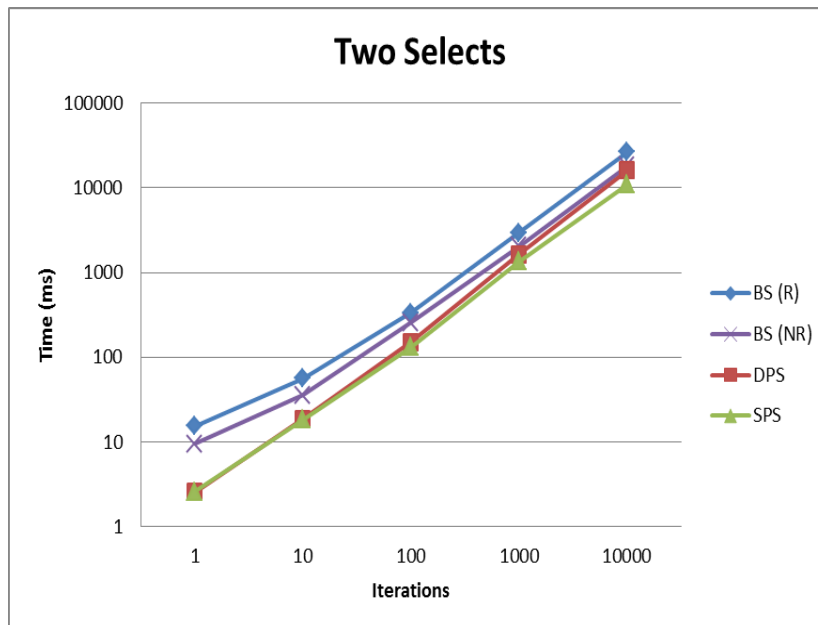


Figure 57. Graph of the execution times in Table 17 for the two selects scenario.

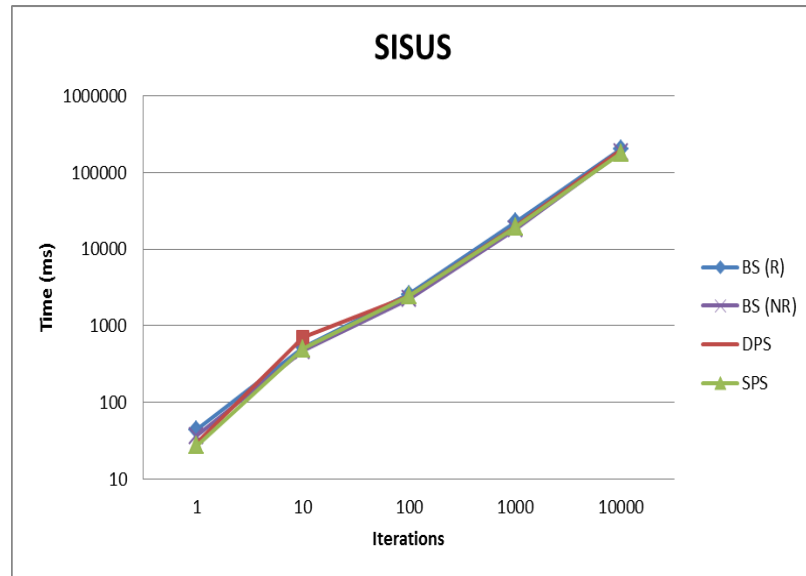


Figure 58. Graph of the execution times in Table 17 for the SISUS scenario.

4.5.4 Results discussion

In this section we will discuss in more depth the results of the initialization and execution tests.

In each table with the results of the initialization tests (section 4.5.2), we can see that the configuration of the Business Manager is the point where S-DRACA spends most of its time initializing. This was expected since it is where the RBAC policies are obtained, the source code generated and compiled. The connection to the Policy Manager is on par with the JDBC connection time and the instantiation time of the Business Manager is neglectable.

When comparing each table we can see in Figure 55 that with JDBC we take a bit more than 10 milliseconds to connect to the database. Regarding S-DRACA, the initialization process takes more than 2 seconds when TFA is not used (around 2.5 to 3 seconds). The overhead of setting up the communication relaying can be calculated as being around 100 milliseconds. The time spent in the case where TFA was used is about 2 times greater when compared to the cases where it is not used (a little over 6 seconds). This can be explained by the fact that TFA introduces more communication overhead, the fact that the CC has to sign a challenge and that the Policy Manager has to validate the signature using the processes already discussed. TFA is clearly a solution that should only be activated when security is imperative. If it is the case, since the time that takes for S-DRACA to initialize is greater than 2 seconds, it is important to let the user know what is happening when it occurs, to avoid user's frustration. It isn't a big problem however, because the initialization is a process that only has to occur once per session.

Regarding the execution tests data shown in Table 17, we can see that using the same prepared statement has better execution times than using different prepared statements. At first this difference is very small, but with each increasing number of iterations it becomes more and more noticeable. This

is an expected result because using the same prepared statement removes the need to keep sending the query and the RDBMS can reuse the execution plan.

Also, using S-DRACA clearly takes more time to execute each statement than using JDBC directly. Looking at the SISUS scenario, the most complex of the three, and the maximum number of iterations, S-DRACA is about 13% slower than the SPS case while relaying the communication through the secure channel and about 3% when not relaying. In the other scenarios it is about 2 to 3 times slower while relaying and 12% to 35% while not. This is most likely because in these scenarios, due only being select statements being executed, the queries are being better optimized by the RDBMS and the S-DRACA overhead becomes more prominent. It is evident that the communication relaying through the secure channel can degrade the performance, but on the other hand it allows the client to use S-DRACA without knowing the database credentials and the data transmission is encrypted. This is an area of S-DRACA that will require further research.

In terms of scalability with load, we can notice in the graphs that by increasing tenfold the number of iterations we get a similar linear increase in execution time for both S-DRACA and JDBC. Hence, data manipulation intensive applications that execute great amounts of queries can expect a similar performance variation to JDBC, and since the policy enforcement is done on the client side, S-DRACA itself also scales with the number of users.

4.5.5 Summary

In this section we evaluated the performance of our solution in terms of the time required to perform some operation on the data. To achieve this we analyzed three different dimensions: the number of iterations, different test scenarios and different methods (i.e. using JDBC directly) to compare with our solution. We showed the results of the tests and then discussed them, concluding that our solution is slower than using JDBC directly, as expected, but followed the same behavior as using only JDBC when the number of iterations increased. Moreover, the relaying of the communication between the client and the database in the Policy Manager can introduce considerable performance degradation.

4.6 Proof of Concept

In this section we will present our proof of concept demonstrating the correct functioning of the implemented components and architecture layers.

The proof of concept, which architecture is shown in Figure 59, uses two applications (a client application that can be used by many users and a controller application that manipulates the policies defined) to demonstrate a sample execution flow. We also implemented a new Policy Extractor to generate the access mechanisms awareness, i.e. the Business Schemas' interfaces.

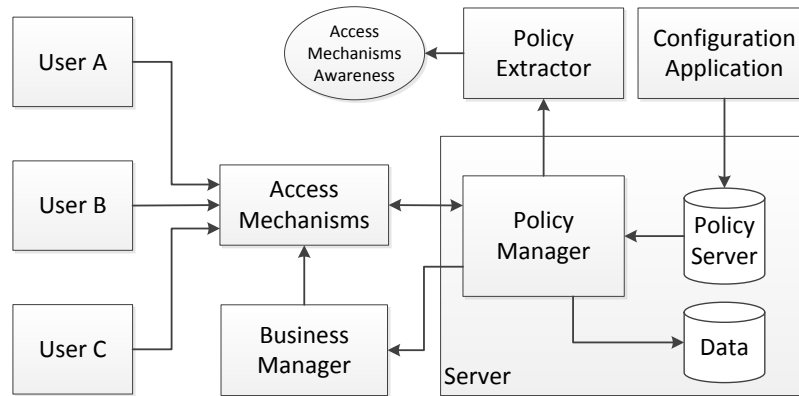


Figure 59. Proof of concept architectural solution.

This section is divided as follows: Section 4.6.1 presents the Policy Extractor tool, section 4.6.2 presents the S-DRACA architecture functioning through the developed applications and section 4.6.3 summarizes the presented content.

4.6.1 Policy Extractor

In this section we present a policy extractor implementation using Java Annotations. The policy extractor is a component that is integral for S-DRACA, as shown in Figure 25. It requests the policies from the policy server through the Policy Manager and generates the interfaces needed by the client application to access the protected data.

There was a Policy Extractor developed in DACA. It was another application that created a Jar file that could be included in the client's application classpath, in order to use the generated interfaces. However, if the policies changed then the Policy Extractor had to be executed again and the new Jar file included. In S-DRACA, we wanted to ease the usage of the Policy Extractor by the client application. We chose to use a custom annotation, not only because it is integrated directly into the application, removing the need for external applications, but also because it is executed every time the client application is compiled, allowing the interfaces to be automatically updated with a simple project recompilation. Figure 60 shows this annotation, named `DACAManagedApplication` (legacy name), which takes the credentials of the user for the Policy Manager (lines 25 and 26), the name of the application (line 29), the Policy Manager network address (line 27 and 28), the authentication method

(line 30), whether the policies are to be updated during compilation (line 31) and if they are to be included in the project (line 32). There are other properties that can be used, such as *keyStorePath* and *keyStorePassword* which allows to use keystores with certificates for SSL/TLS.

```

24  @DACAManagedApplication(
25      username = "user1",
26      password = "guest",
27      url = "localhost",
28      port = 9000,
29      app = "App",
30      authenticationMethod = AuthenticationMethod.PSKSSL,
31      updatePolicies = true,
32      includePolicies = true
33  )
public class Login extends javax.swing.JFrame {

```

Figure 60. Policy Extractor annotation.

With this information the Annotation Processor that was developed can authenticate with the Policy Manager, request the policies from the policy server, implement the interfaces and add them to the project. The *updatePolicies* parameter on line 31 allows the developers to prevent the policies from being updated during compilation, saving time in exchange. The *includePolicies* parameter on line 32 can disable the annotation if for some reason another tool is used. These parameters have the default value set to *true*.

Figure 61 shows the declaration of the developed Annotation Processor, called “DMAAP”. We can notice on line 33 that it supports our annotation “DACAManagedApplication” and that the supported Java version is the release 8 (line 34). For the class to be successfully used as an Annotation Processor it extends the abstract class *AbstractProcessor* (line 35).

```

33  @SupportedAnnotationTypes("annotation.DACAManagedApplication")
34  @SupportedSourceVersion(SourceVersion.RELEASE_8)
35  public class DMAAP extends AbstractProcessor {

```

Figure 61. Annotation processor declaration.

Figure 62 shows the signature of the *process()* method declared in the *AbstractProcessor* class. This method is called during the compilation process and provides two parameters: the annotations variable, which is a list of the annotations that correspond to the supported types and a *RoundEnvironment* variable which contains information about the round of annotation processing, such as errors or the annotations being processed. The returned boolean value indicates whether this annotation processor takes ownership over the set of annotations passed. By claiming ownership we indicate that the annotations are ours and that we processed them. There is also a protected variable, called *processingEnv* that allows to create source code, print messages and provides some other functionalities.

```

43  @Override
44  public boolean process(Set<? extends TypeElement> annotations,
45                      RoundEnvironment roundEnv) {

```

Figure 62. Annotation processor's process method signature.

Figure 63 shows the main generation process of the security data structures and the Business Schemas' interfaces. First, the list of Business Schemas is obtained from the Policy Server using the protocol detailed in section 4.1.4 (line 68). Then we create a source file using the *processingEnv* variable with the name of the Business Schema (line 71 and 72). If the *includePolicies* attribute in the annotation was set to true (line 74), then we request the generation of the Business Schema's interface to the *InterfaceGenerator* class and write the output into the source file (line 77 and 78).

```

68      List<Class> interfaces = getInterfaces(ann, orginfo, orgActive);
69
70      for (Class i : interfaces) {
71          JavaFileObject jfo = processingEnv.getFiler().createSourceFile(
72              i.getPackage().getName() + "." + i.getSimpleName());
73
74          if (ann.includePolicies()) {
75              //Add the content to the newly generated file.
76              try (PrintWriter pw = new PrintWriter(jfo.openWriter())) {
77                  pw.println(InterfaceGenerator.genInterfaceSourceCode(
78                      i, interfaces, orginfo, orgActive));
79                  pw.flush();
80              }
81          }
82      }

```

Figure 63. The Business Schemas interfaces generation process.

Figure 64 shows the *genInterfaceSourceCode()* method in the *InterfaceGenerator* class. It generates the source code for the interfaces and follows the basic structure of a Java class, starting with the package declaration (line 29) and finishing with the methods used for the sequence lifecycle (line 35). The generation process itself uses Java Reflection to determine the declared methods of the target class. The parameter *c* is a class reference for which an interface must be generated. The *otherClasses* parameter is a list of the other classes that are also being processed and it's useful when the class being processed imports other generated classes, such as the security data structure. The *seqInfo* parameter has the defined sequences of Business Schemas, used to generate the "next" methods, and the *seqActive* parameter indicates if the sequence control mechanism is active or not.

```

26      public static String genInterfaceSourceCode(Class c, List<Class> otherClasses,
27          Map<Integer, Map<Integer, String>> seqInfo, boolean seqActive) {
28          StringBuilder isc = new StringBuilder();
29          getPackage(isc, c);
30          getImportsForChoreography(isc, c, seqInfo);
31          getImportsForClass(isc, c, otherClasses);
32          getInterfaceDeclaration(isc, c, otherClasses);
33          getInterfaceFields(isc, c, otherClasses, seqInfo, seqActive);
34          getMethods(isc, c, otherClasses);
35          generateNextMethods(isc, c, seqInfo);
36          isc.append("\n");
37          return isc.toString();
38      }

```

Figure 64. InterfaceGenerator main method.

The output of the Annotation Processor can be seen in the IDE with the appearance of the “Generated Sources (ap-source-output)” folder (see Figure 65). The “BusinessInterfaces” folder contains the common interfaces to all Business Schemas, the “Classes” folder contains the data structures associated with the roles (see Figure 39) and the rest of the folders have the Business Schemas usable by the application.

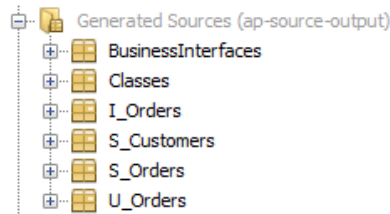


Figure 65. Generated sources folder.

4.6.2 Sample applications

This section will present the sample applications developed to demonstrate the correct functioning of S-DRACA. Two sample applications were developed for this purpose: a client application that uses the Policy Extractor presented, which allows a user to perform some high level operations and an application to easily change the defined policies. The focus of this proof of concept will be around the sequence controller since it was the new functional component developed, but we will show the client application also becoming aware of changes made to the user’s roles automatically.

Figure 66 shows the initial window of the configurator that allows to change the defined policies in the policy server.

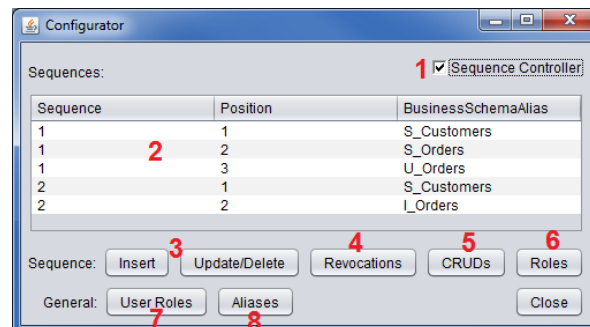


Figure 66. Initial window of the configurator.

The functionalities are represented on the window with the numbers in red. 1) The status of the sequence controller, which can be either enabled or disabled. 2) A table with the current sequences in effect, which shows each sequence identifier and the authorized business schema for each position. 3) The insert and update/delete buttons allow to insert, update and delete sequences. 4) The Revocations button allows to see the current revocations for each sequence as well as insert and delete them. 5) The CRUDs button allows to check the allowed CRUDs for each Business Schema in a sequence position. 6) The Roles button allows to check and update the roles associated with each sequence. 7) The User Roles button allows to check and update the roles associated with each sequence. 8) The Aliases button allows to check and update the aliases associated with each sequence.

The User Roles button allows to see and update the current roles associated with each user. Finally, 8) allows to see and update the aliases defined for each Business Schema that are used in the sequences.

The client application starts by showing a generic login window which requests the username and password. It then proceeds by using one of the developed authentication mechanisms and enabling data encryption (using the PSK-SSL solution). After the user logs in the client application shows a window with the customers present in the Northwind sample database (see Figure 67).

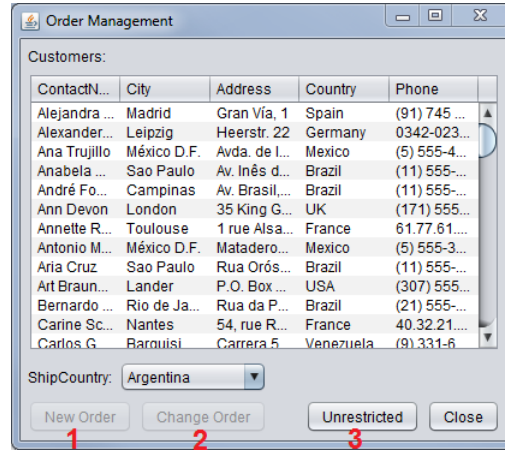


Figure 67. Client application showing the list of customers.

Figure 68 shows how the client application fills the table in Figure 67 with the information using S-DRACA. First, a Business Schema that handles the select expressions on the Customers table is instantiated with the select all CRUD expression (line 260). Then, the CRUD expression is executed (line 261) and for each row in the Business Schema's LDS (line 262) a line in the graphical user's interface table is added through the default table model (lines 263 to 265).

```

260     Cust = man.instantiateBS(s_customers, s_customers_S_Customers_all, session);
261     Cust.execute();
262     while (Cust.moveToNext()) {
263         dtm.addRow(new Object[]{
264             Cust.ContactName(), Cust.City(), Cust.Address(), Cust.Country(), Cust.Phone()
265         });
266     }

```

Figure 68. Code that adds information into the Customers' table.

The client application implements the two sequences shown in Figure 66 via the “New Order” button (number 1 in the figure), which allows to create a new order (sequence 2) and the “Change Order” button (number 2 in the figure), which allows to modify an existing order (sequence 1). The “Unrestricted” button (number 3 in the figure) shows a test window to try to instantiate and execute any Business Schema in any order, which allows to evaluate the complete correctness of S-DRACA. We will update an existing order, where each Business Schema in the sequence is needed by the one following it. We start with the correct revocations applied, i.e. with the first Business Schema revoked only in the third position, and then we try it again with the first Business Schema revoked in the second position, which should prevent the client application from viewing the list of orders of a

customer. We will finally revoke the current role of the user and see the Business Schemas disappear from the “Unrestricted” window.

Figure 69 shows the client application after having selected a customer and pressed the “Change Order” button, which shows a list of orders obtained from the “S_Orders” Business Schema, for the selected customer. After selecting an order to update and pressing the “Update Order” button in the window, a new window appears (see Figure 70) that allows to change some fields of the order. After pressing the “Update” button and if the Business Schema “U_Orders” can be executed, the confirmation dialog appears indicating the success of the update.

We then revoked the first Business Schema in the second position of the sequence in the Configurator. This will prevent the client application from being able to retrieve the orders placed by a customer using the sequence 1 (see Figure 71), creating an error dialog if the user tries to perform the second step in the update order sequence.

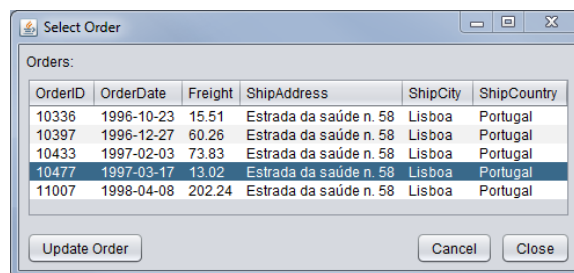


Figure 69. Second step in updating an order.

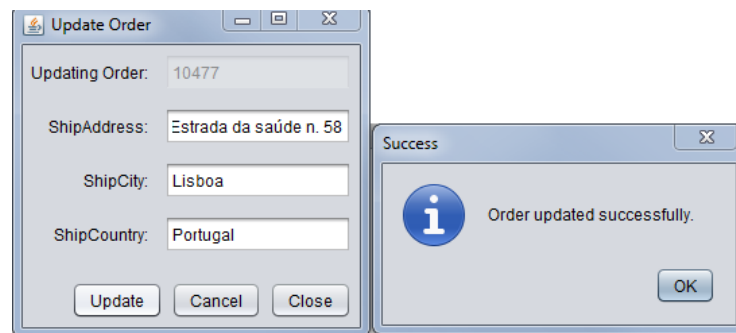


Figure 70. The final update step and confirmation dialog.

Then we tried to execute the same operation of updating an order of a customer. As expected, when we now tried to retrieve the orders of a customer to update, an error message appeared, indicating that the operation is not authorized (see Figure 72). To see exactly what is happening we can see Figure 73, which shows the unrestricted window. On (1) it shows the current roles of the user, (2) shows the Business Schemas associated with the roles and (3) will attempt to instantiate them. If they can be instantiated an “Authorized” result shows and the Business Schema goes to area (5). Area (4) shows the state of the sequence controller and the last time the policies were updated. The “Try Execute” button on area (6) allows to try to execute a selected Business Schema from area (5), the result of which appears under it. The “New Sequence” button on area (7) clears the instantiation

history for a new sequence. Note that after the instantiation of the “S_Orders” Business Schema, the “S_Customers” can no longer be executed as shown by the “Not Authorized” message.

To demonstrate that the client applications are aware of the changes made to the policies using S-DRACA we used the Configurator to revoke the role of the user (see Figure 74). After that we can see in Figure 75 that the role disappeared from area (1) and no Business Schemas are available.

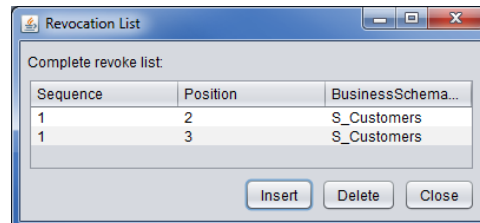


Figure 71. Configurator revocation list window, after revoking the first Business Schema in the second position.

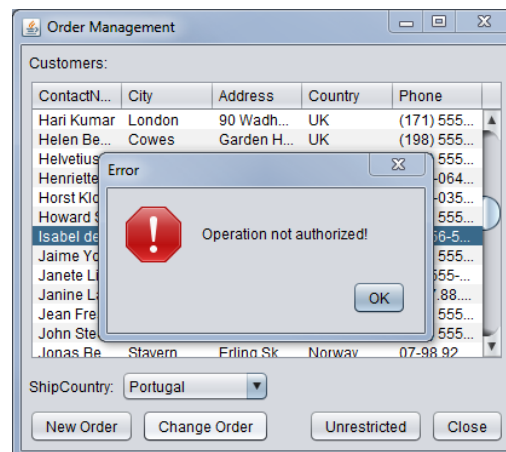


Figure 72. Client application's error message for trying to execute a revoked Business Schema.

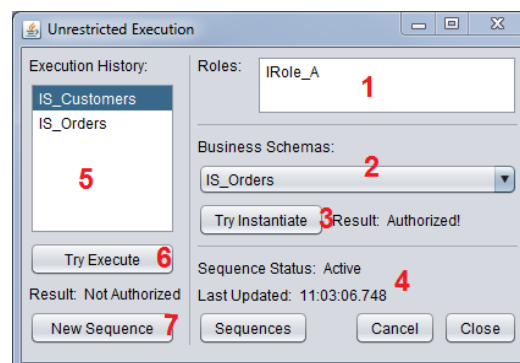


Figure 73. The client's unrestricted window.

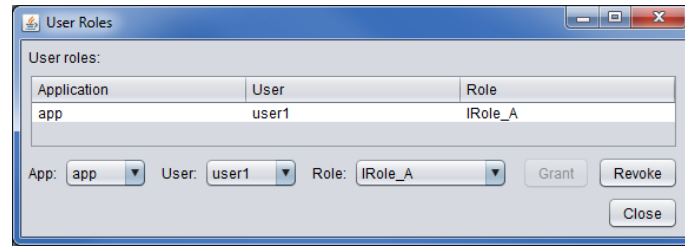


Figure 74. The Configurator's user roles windows.

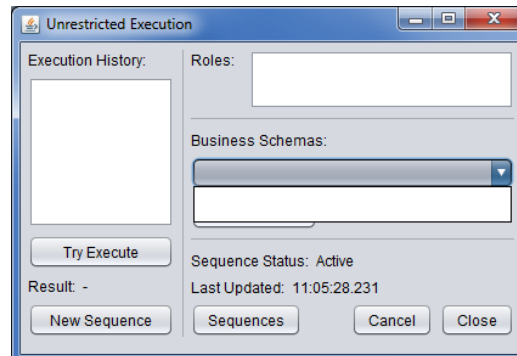


Figure 75. Client's unrestricted window after the user had its roles revoked.

The client application is able to update its graphical user interface because the Business Manager can register a listener (with the legacy name “DACChangeListener”, shown in Figure 76). This listener can be used by the client application to be notified when changes to the sequences (lines 11 and 12) or the policies (line 13) are made and adjust the interface accordingly, e.g. update a combobox as shown in Figure 77.

```

10 public interface DACChangeListener {
11     public void sequenceStatusChanged(boolean status);
12     public void sequenceChanged(Map<Integer, Map<Integer, String>> sequences);
13     public void policiesChanged(Map<String, HashMap<Integer, String>> businessSchemas);
14 }

```

Figure 76. The change listener interface.

```

449 @Override
450 public void policiesChanged(Map<String, HashMap<Integer, String>> businessSchemas) {
451     jComboBox1.removeAllItems();
452     businessSchemas.keySet().stream().forEach((bs) -> {
453         jComboBox1.addItem(bs.split("[.]" ) [1]);
454     });
455 }

```

Figure 77. Operation performed when policies change.

The client application is not required to use the listener. However, it can be used to prevent runtime errors when some policy is changed by disabling a previously authorized operation that is no longer authorized.

4.6.3 Summary

In this section we presented a policy extractor that uses metadata added to the client application using a custom Java Annotation. From the metadata the Java Annotation Processor was capable of generating the required interfaces for the client application to access the data stored in the database with. Then we presented two applications: a configurator that allowed to easily manipulate the defined policies in the Policy Server and a client application that showed that it is possible to enforce the sequence of Business Schemas on the client side, changing its behavior automatically when the policies in the Policy Server are modified during runtime.

5 Conclusion

In this section we will discuss the work done, some identified problems and how S-DRACA can evolve. We will also study the applicability of S-DRACA for other data stores including in the big data scenario. This section is divided as follows: Section 5.1 will discuss the results of the sequence controller and the next steps, section 5.2 will discuss the effectiveness and the problems of the security layer as well as what to do to enhance it and section 5.3 will study the possible applicability of S-DRACA with other access control policies and in big data scenarios and present the future work.

5.1 Sequence controller

The developed sequence controller was an attempt to provide greater control to security experts in terms of how and when the client applications can use the authorized CRUD expressions.

The first iteration developed in S-DRACA enables the security experts to do that, but it is too limited. The sequences can only follow a single path of the general digraph (see Figure 28 and Figure 29). Furthermore, the current extension to the RBAC model only allows to define, in a sequence's lifecycle, which Business Schemas are revoked at each step. The ideal solution would allow a more expressive digraph to be used directly (like the one shown in Figure 28). Since a sequence could take several different paths, the digraph's lifecycle would also support the definition of which next Business Schema are available at a given position, depending on the path taken so far.

The reason for this possible evolution of the sequence controller is due to the fact that one operation might not always require the exact same steps. Consider a web form that allows users to login to some area in a website. If the credentials used are correct in the first three attempts, the user logins successfully, but if the user fails to provide the correct credentials three times in a row the login sequence now requires the user to answer a security question to unlock the account. In this case the same login sequence might take two paths depending on how many times the user fails to provide the correct credentials.

5.2 Security layer

In this section we will discuss the goals achieved by the security layer developed, its problems and how it can be enhanced.

First, there were several authentication mechanisms for users implemented, each with a different security/communication overhead. The two simplest, using a hash-based password and the challenge-response mechanisms, since they are implemented using standard algorithms, we feel that they achieved what they were designed for. On the other authentication mechanisms using a shared secret together with SSL/TLS, we were only able to implement one of them and it uses reflection so the abstraction of the Java SSL sockets is lost. The best way to implement this would require the

implementation of the SSL/TLS protocol to support natively the EKE mechanisms. In this case all three solutions presented could be implemented.

The goal of using the SSL/TLS channel, created during authentication, is to encrypt the data communication and to prevent the client applications from requiring the database credentials was a partially successful. We did achieve our goal, but we also use Java Reflection, which makes us dependent on the current implementation of the SQL Server connection objects for JDBC. Furthermore, there can be some performance degradation in the process of relaying the communication through the secure channel.

The attempt to hide the queries being executed from the client applications so that users cannot gain access to the database schema has been successful. Using the RemoteCall stored procedure and session identifiers we were able to execute any CRUD expression defined in the policy server and the client applications only have access to the identifiers.

Finally, there is still a security vulnerability that was not addressed entirely with this security layer. An intruder that is able to get access to a client application, using Java Reflection, is able to obtain the Connection object that allows our solution to communicate with the database. It is not possible to prevent an intruder with this object from executing any CRUD expression that he wants in the client side.

This problem is partially solvable using the relaying of the communication between the client application and the database in the Policy Manager, to hide the actual credentials used to access the database. Additionally, the real credentials can be used to access an account that is able to execute just stored procedures, this way restricting an intruder to execute only the defined stored procedures. This solution, however, still allows the intruder to bypass the sequence controller and even the RBAC policies if the stored procedures can be executed by anyone.

A better solution would involve the generation of the Business Schemas' implementation on a proxy server, not accessible by the users and the implementation of the Business Schemas in the client side that would communicate with this proxy server to execute the required methods. This way the client application would no longer possess a connection object for intruders to use and the relaying of the communication between the proxy server and the database would only be required if the proxy server is not on the same machine as the database, or else an intruder could capture the credentials by capturing the network packets.

5.3 Discussion

In this section we will discuss how to integrate S-DRACA with other data sources and the future work we intend to do.

This section is divided as follows: Section 5.3.1 will discuss how S-DRACA can be applied to other access control policies, section 5.3.2 will discuss how S-DRACA can be applied to other data sources, including some work done in this direction, and section 5.3.3 will discuss the future work.

5.3.1 Application to other access control policies

In this section we will discuss the applicability of other access control policies in S-DRACA, such as MAC, DAC and ABAC. We have shown, in section 4.2, how an extension to RBAC can be used to define the actions a user can perform. We associate sequences of Business Schemas to roles and a user that plays a given role can execute the associated sequences.

Using MAC as the underlying access control policy, we could associate sequences of Business Schemas with security levels and a set of categories. This way, only a user with clearance to execute a sequence would be allowed to do so. We could also apply the security based MAC or the integrity based MAC, presented in section 2.1.2, considering the execution of a sequence of Business Schemas can be considered as reading them and their modification by security experts as writing on them.

Regarding DAC, we don't find it to be very adequate when integrated with S-DRACA, mainly because its strongest points reside in its ability to provide the subjects, with certain permissions on an object, to pass those permissions to other subjects. In our case the objects are the sequences of Business Schemas, which we don't want to allow the subjects to create and pass their permission to execute it to others.

Considering ABAC, we could associate with each sequence of Business Schemas a set of attributes that the subjects must possess in order to be able to execute them. For example, a particular sequence may have attributes that allows subjects to execute them only on certain periods of time. Furthermore, when a subject attribute changes, no other changes are required on the objects' attributes to automatically update the authorizations.

Although S-DRACA was based primarily on RBAC, inheriting the model used in DACA, nothing prevents other access control policies from being used, except for the method of obtaining the list of authorized operations when a Business Manager is initializing, which must take them into account.

5.3.2 Integration with other data sources

In this section we will discuss the applicability of S-DRACA to other data sources, namely Hadoop. We will also briefly introduce Hive (Apache, n.d.-a) and HBase ("HBase," n.d.) that runs on top of Hadoop (Apache, n.d.-b) and the Hadoop Distributed File System (HDFS)(Apache, n.d.-c). HDFS, as the name implies, is a distributed file system and Hadoop is a framework that allows performing distributed processing on the data stored in the HDFS. These tools do not understand SQL and it isn't possible to connect to Hadoop using JDBC directly, which means that S-DRACA cannot be used, since the Business Schemas depend on JDBC to perform the authorized operations.

To integrate our solution we need to use tools like Hive that enables users to query data on the Hadoop/HDFS through a SQL-like language, by translating the queries into the jobs that can be understood by Hadoop. If Hive can receive SQL-like statements then, if we were able to use JDBC to connect to a Hive endpoint, we would only be required to change how the Business Schemas are

implemented so that they use the new SQL-like language instead of the standard SQL language used by relational databases.

We will also study the applicability of S-DRACA to control the access to data stored in HBase. HBase is a distributed, column-oriented database that runs on top of Hadoop/HDFS that adds functionalities that Hive lacks. Since S-DRACA requires a JDBC connection to function, one could use a tool like Phoenix (“Developing with Phoenix,” n.d.) to connect to an HBase node using JDBC.

Integrating S-DRACA into the big data scenario required some changes from the previous implementation of the Business Schema’s generator, responsible for the step 4 in Figure 25. It was a monolithic class that generated the Business Schemas for SQL Server (which supported any other database as long as the JDBC operations remained the same). We changed its design so that implementation dependent information was delegated to the specific implementations of the generator, hence the general generator (BSGenerator) became an abstract class that implemented the *generate()* and *compile()* procedures that are common to all implementations and delegated the generation of the imports, class declaration, variables, and other required methods to the implementing classes. We noticed that some implementing classes might require additional libraries, so we also delegate the definition of the classpath to each one of them. With this approach we only have to change the generator we want and the Business Schemas will be implemented for a different database technology.

To ease the development of a new implementation, an object is provided by the abstract generator that contains context information about the Business Schema being implemented, such as its type (a select, update, insert or delete), its name, and other relevant information.

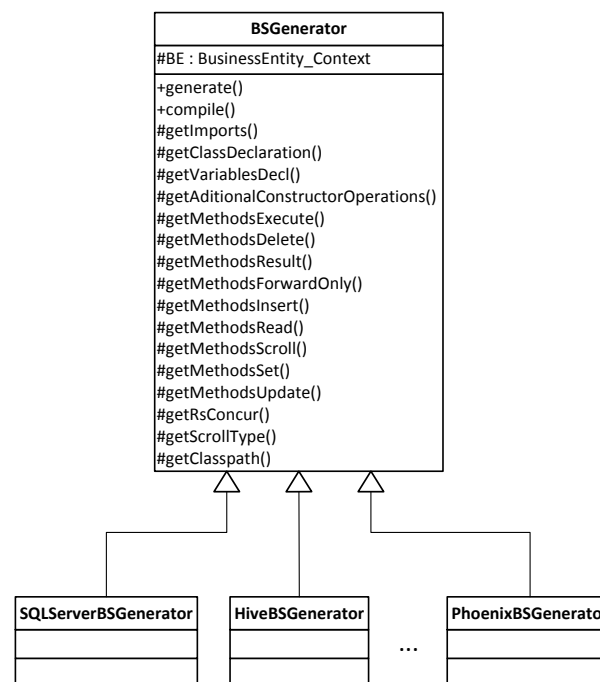


Figure 78. New Business Schema generator model.

Implementing a generator for Hive is not as straightforward as implementing the SQL Server generator (it only forwards the calls to the result set and the direct access modes are all directly supported), mainly because the insert, update and delete statements are not supported by the JDBC driver on either direct and indirect access modes, due to the append-only nature of HDFS, meaning that Hive allows only to append (i.e. insert into) and overwrite (i.e. insert overwrite) the data from one table into another. Using Phoenix and HBase eases the problem, by supporting insertion/update and deletion which Phoenix maps into the upsert (updates or inserts) and delete commands. However, the indirect access mode is still not supported. We will first analyze how to implement the generator for Hive, because it requires additional work since it isn't possible to insert values into a table from the query directly, only from another table. The Phoenix generator was not implemented, but we discuss the major differences that it has from the Hive generator. Note that it is assumed that the queries configured in the Policy Server are written for the target database technology and that the Policy Server remains in a SQL Server instance.

5.3.2.1 Integration with Hive

In this section we will present the implementation details of a generator for Hadoop using Hive. The Hive generator requires some work in both direct and indirect modes, but they can be merged together if we consider that: in the direct access mode all fields to update/insert are provided all at once and that in the indirect mode they are provided one at the time, between a *beginInsert()* and a *endInsert()* method call. The delete operation deletes the current row in the indirect access mode and a single row in the direct access mode. The select statement does not require any additional work.

To support the insert and update statements, the generator creates a data structure, dependent from the schema of the table being updated or inserted. In the direct access mode, an insert statement has to provide a value for every column in the table in order, which allows the structure to be filled and then used to insert a new row into the table. In the indirect access mode, each insert method must be called in order to fill the structure. The insert operation then requires the values of the columns to be sent to an application in the server running Hive that stores the values in file inside the HDFS, load the file into a new table and then append the temporary table into the target table. The loading of the file into a temporary table and the appending operation are supported by Hive.

The delete operation requires, in the direct access mode, to determine the row to be deleted. Hence, the delete statement, which must index the row by its key, has to be parsed. Then an insert overwrite command is used that inserts all the data except that row back into the table. Because the insert overwrite replaces the entire table with the data inserted, only the row we wanted to delete is removed. In the indirect access mode, because the current row is known, including its row key, the delete operation requires the same insert overwrite operation.

The update operation requires, in the direct access mode, to determine the row being updated. This requires the query to be parsed to determine the table and the row. When using the direct access mode all columns must be updated, in order. In the indirect access mode, since we know the values of

the current row's columns, we support updates on a subset of the columns. To update the table we are again required to delete the current row in the table and to insert the new row with the same key, so the delete and insert operations are reused, meaning this is the most expensive operation to perform.

We also extrapolated a generator for HBase and Phoenix. Since it supports inserts and updates through the upsert statement and deletes through the delete statement in the direct access mode, it only requires modifications in the indirect access mode.

To support the indirect access mode we could use the same internal data structure as the one used for the Hive generator. For the insert operation, every column must be assigned a new value, and for the update operation the values of the current row are used, meaning that just some columns may be updated. The Business Schema then executes an upsert statement to update/insert the data depending whether the row key exists or not. The delete operation is easily supported by executing a delete statement which is supported by Phoenix.

5.3.2.2 The insertion conflict

It is important to note that neither Phoenix nor Hive have support for full transactions (just at the row level for updates in HBase), and that auto-incrementing row keys are not supported either by Hadoop or HBase. This means that to insert a new row we must have a row key that isn't being used. We could select the max row key from the table where we want to insert a new row and increment it manually, but nothing prevents another application from doing the same since there is no full transaction support, i.e. two applications can read the same maximum row key value K and then one of them inserts the new row with the row key $K+1$. Then the other application also increments K and since $K+1$ already exists the upsert statement will update the column values, overwriting the first insertion.

One possible solution can make use of a centralized server that performs the insert operations. This way it knows the next row key to use for each table.

5.3.3 Future work

In this section we will discuss the possible future work for the work done with this dissertation. The future work can take many directions in every layer that was developed or extended.

In the sequence controller component, it can be too restrictive for the security expert to define only paths with a single direction (i.e. go forward to the next Business Schema). We intend to continue expanding this component so that it becomes possible to define paths with other branching paths, so that a security expert can define sequences that under some condition can change.

In the security layer, we still have some authentication mechanisms that we wanted to implement but we were unable to do so, because the used programming language (Java) does not allow to extend the implementation of the communication sockets, namely the SPEKE and SRP mechanisms. Also, we want to address the problem regarding the usage of Java Reflection to obtain the Connection object that allows an intruder to access the database. Some solutions have been presented in section 5.2.

Regarding the configurability of the extended RBAC model in the Policy Server, we have a simple application that allows to configure it from Java interfaces that define the authorized operations on each Business Schema, but it only configures statically the sequences and associated metadata. Some work will be performed so that we create a more powerful configurator to ease the configuration process and even automate what can be automated.

6 References

- Abramov, J., Anson, O., Dahan, M., Shoval, P., & Sturm, A. (2012). A methodology for integrating access control policies within database development. *Computers & Security*, 31(3), 299–314. doi:10.1016/j.cose.2012.01.004
- Abramowitz, M., & Stegun, I. A. (1972). Primitive Roots. In *Handbook of Mathematical Functions: with Formulas, Graphs, and Mathematical Tables* (9th ed., p. 827). New York, New York, USA: Dover.
- Agrawal, R., Kiernan, J., Srikant, R., & Xu, Y. (2002). Hippocratic databases. *Proceedings of the 28th ...*, 4(1890). Retrieved from <http://dl.acm.org/citation.cfm?id=1287383>
- Ahn, G., & Hu, H. (2007). Towards realizing a formal RBAC model in real systems. *Proceedings of the 12th ACM symposium on Access ...*, 215. doi:10.1145/1266840.1266875
- Apache. (n.d.-a). Apache Hive. Retrieved May 27, 2014, from <https://hive.apache.org/>
- Apache. (n.d.-b). Introduction to Hadoop. Retrieved May 27, 2014, from <https://hadoop.apache.org/>
- Apache. (n.d.-c). HDFS Architecture. Retrieved May 27, 2014, from https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
- Ashley, P., Hada, S., & Karjoth, G. (2003). *W3C Enterprise privacy authorization language (EPAL 1.2). W3C Member Submission* (pp. 1–58). doi:10.3109/02699206.2011.561398
- Barka, E., & Sandhu, R. (2000). A role-based delegation model and some extensions. *23rd National Information Systems Security ...*. Retrieved from <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.26.4593&rep=rep1&type=pdf>
- Bell, D., & LaPadula, L. (1973). Secure Computer Systems: A Mathematical Model. Volume II., II(May 1973). Retrieved from <http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=AD0771543>
- Bellovin, S. M., & Merritt, M. (n.d.). Cryptographic protocol for secure communications. Retrieved from <https://www.google.com/patents/US5241599>
- Bellovin, S., & Merritt, M. (1992). Encrypted key exchange: Password-based protocols secure against dictionary attacks. *Research in Security and Privacy, 1992 ...*. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=213269
- Bellovin, S., & Merritt, M. (1993). Augmented encrypted key exchange: a password-based protocol secure against dictionary attacks and password file compromise. *... on Computer and communications security*. Retrieved from <http://dl.acm.org/citation.cfm?id=168618>
- Biba, K. (1977). Integrity considerations for secure computer systems. Retrieved from <http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA039324>
- Biham, E., Anderson, R., & Knudsen, L. (1998). Serpent: A new block cipher proposal. *Fast Software Encryption*. Retrieved from http://link.springer.com/chapter/10.1007/3-540-69710-1_15

- Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), 422–426. doi:10.1145/362686.362692
- Bonner, A. (1997). Transaction Datalog: a compositional language for transaction programming. *Database Programming Languages*, 1–30. Retrieved from http://link.springer.com/chapter/10.1007/3-540-64823-2_21
- Borders, K., Zhao, X., & Prakash, A. (2005). CPOL: High-performance policy evaluation. *Proceedings of the 12th ACM conference* Retrieved from <http://dl.acm.org/citation.cfm?id=1102142>
- Caires, L., Pérez, J., & Seco, J. (2011). Type-based access control in data-centric systems. ... *Languages and Systems*. Retrieved from http://link.springer.com/chapter/10.1007/978-3-642-19718-5_8
- Canfora, G., Visaggio, C. A., & Paradiso, V. (2009). A Test Framework for Assessing Effectiveness of the Data Privacy Policy's Implementation into Relational Databases. *2009 International Conference on Availability, Reliability and Security*, 240–247. doi:10.1109/ARES.2009.153
- Cartão de Cidadão. (n.d.). Retrieved April 29, 2014, from <http://www.cartaodecidadao.pt/index.php?lang=en.html>
- Chamberlin, D., & Boyce, R. (1974). SEQUEL: A structured English query language. *Proceedings of the 1974 ACM SIGFIDET (....* Retrieved from <http://dl.acm.org/citation.cfm?id=811515>
- Chaudhuri, S., Dutta, T., & Sudarshan, S. (2007). Fine grained authorization through predicated grants. In *Proceedings - International Conference on Data Engineering* (pp. 1174–1183). doi:10.1109/ICDE.2007.368976
- Chlipala, A. (2010). Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications. *Proceedings of the Ninth USENIX Symposium on Operating Systems Design and Implementation*, 105–118. Retrieved from http://www.usenix.org/events/osdi10/tech/full_papers/Chlipala.pdf
- Codd, E. F. (1970). A relational model of data for large shared data banks. 1970. *M.D. computing : computers in medical practice*, 15(3), 162–6. Retrieved from <http://www.ncbi.nlm.nih.gov/pubmed/9617087>
- CoherentPaaS. (n.d.). Retrieved June 17, 2014, from <http://coherentpaas.eu/>
- Cooper, E., Lindley, S., Wadler, P., & Yallop, J. (2007). Links: Web programming without tiers. *Formal Methods for Components* Retrieved from http://link.springer.com/chapter/10.1007/978-3-540-74792-5_12
- Corcoran, B., Swamy, N., & Hicks, M. (2009). Cross-tier, label-based security enforcement for web applications. *Proceedings of the 2009 ACM* Retrieved from <http://dl.acm.org/citation.cfm?id=1559875>
- Cranor, L., Langheinrich, M., Marchiori, M., Presler-Marshall, M., & Reagle, J. (2002). The Platform for Privacy Preferences 1.0 (P3P1.0) Specification. *W3C*, 0, 1–76. Retrieved from <http://www.w3.org/TR/P3P/>

- Daemen, J., Rijmen, V., & Leuven, K. U. (1999). AES Proposal : Rijndael. *Complexity*, 1–45. Retrieved from <http://ftp.csci.csusb.edu/ykarant/courses/w2005/csci531/papers/Rijndael.pdf>
- Developing with Phoenix. (n.d.). Retrieved May 27, 2014, from <https://phoenix.incubator.apache.org/>
- Eclipse. (2008). EclipseLink. Retrieved April 28, 2014, from <https://www.eclipse.org/eclipselink/>
- Ferraiolo, D., & Kuhn, D. (1992). Role-based access controls. *15th National Computer Security Conference*, 554–563. Retrieved from <http://arxiv.org/abs/0903.2171>
- Fischer, J., Marino, D., Majumdar, R., & Millstein, T. (2009). 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, pp. 173–194). doi:10.1007/978-3-642-03013-0_9
- Halfond, W., Viegas, J., & Orso, A. (2006). A classification of SQL-injection attacks and countermeasures. *Proceedings of the IEEE* Retrieved from <http://www.cc.gatech.edu/fac/Alex.Orso/papers/halfond.viegas.orso.ISSSE06.pdf>
- HBase. (n.d.). Retrieved May 27, 2014, from <https://hbase.apache.org/>
- Hu, V., Ferraiolo, D., & Kuhn, R. (2014). Guide to Attribute Based Access Control (ABAC) Definition and Considerations. *NIST Special* Retrieved from <http://nvlpubs.nist.gov/nistpubs/specialpublications/NIST.sp.800-162.pdf>
- IETF. (n.d.-a). RFC 2560: X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. Retrieved April 05, 2014, from <http://www.ietf.org/rfc/rfc2560.txt>
- IETF. (n.d.-b). RFC 6101: The Secure Sockets Layer (SSL) Protocol Version 3.0. Retrieved from <http://tools.ietf.org/html/rfc6101>
- IETF. (n.d.-c). RFC 5246: The Transport Layer Security (TLS) Protocol - Version 1.2. Retrieved from <http://tools.ietf.org/html/rfc5246>
- Jablon, D. P. (n.d.). Cryptographic methods for remote authentication. Retrieved from <https://www.google.com/patents/US6226383>
- Jayaraman, K. . K., Tripunitara, M. M. ., Ganesh, V. V. ., Rinard, M. . M., & Chapin, S. . S. (2013). MOHAWK: Abstraction-refinement and bound-estimation for verifying access control policies. *ACM Transactions on Information and System Security*, 15(4), 1–28. Retrieved from <http://www.scopus.com/inward/record.url?eid=2-s2.0-84878578416&partnerID=40&md5=e6515eda5edc27d305415bd07cb79a77>
- JBOSS. (n.d.). What is Object/Relational Mapping? Retrieved March 03, 2014, from <http://hibernate.org/orm/what-is-an-orm/>
- JBOSS. (2001). Hibernate. Retrieved March 03, 2014, from <http://docs.jboss.org/hibernate/orm/4.1/quickstart/en-US/html/>
- Kaufman, C., Perlman, R., & Speciner, M. (2002a). Hashes and Message Digests. In *Network Security: Private Communication in a PUBLIC World* (2nd ed., pp. 117–143). Prentice Hall.

- Kaufman, C., Perlman, R., & Speciner, M. (2002b). *Network Security: Private Communication in a PUBLIC World. Network Security: Private Communication in a PUBLIC World* (Second., pp. 215–301, 477–497). Prentice Hall.
- Kaufman, C., Perlman, R., & Speciner, M. (2002c). Modes of Operation. In *Network Security: Private Communication in a PUBLIC World* (2nd ed., pp. 95–114). Prentice Hall.
- Komlenovic, M., Tripunitara, M., & Zitouni, T. (2011). An empirical assessment of approaches to distributed enforcement in role-based access control (RBAC). *Access*, 121–132. doi:10.1145/1943513.1943530
- Lampson, B. W. (1974). Protection. *ACM SIGOPS Operating Systems Review*. doi:10.1145/775265.775268
- LeFevre, K., & Agrawal, R. (2004). Limiting disclosure in hippocratic databases. *Proceedings of the ...*, 108–119. Retrieved from <http://dl.acm.org/citation.cfm?id=1316701>
- Malenfant, J., Jacques, M., & Demers, F. (1996). A tutorial on behavioral reflection and its implementation. *Proceedings of the Reflection*. Retrieved from <http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/malenfant/malenfant.pdf>
- Microsoft. (n.d.). ADO.Net.
- Microsoft. (2007). LINQ (Language-Integrated Query). Retrieved March 03, 2014, from <http://msdn.microsoft.com/en-us/library/bb397926.aspx>
- Microsoft. (2012). Windows Server 2012 R2 Preview.
- Morin, B., Mouelhi, T., Fleurey, F., Le Traon, Y., Barais, O., & Jézéquel, J.-M. (2010). Security-driven model-based dynamic adaptation. *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*, 205. doi:10.1145/1858996.1859040
- National Bureau Of Standards. (1999). Data Encryption Standard (DES). *Technology*, 46-3, 1–26. Retrieved from <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>
- NIST. (n.d.). ATTRIBUTE BASED ACCESS CONTROL (ABAC) - OVERVIEW. Retrieved May 17, 2014, from <http://csrc.nist.gov/projects/abac/>
- OASIS. (n.d.). WS-BPEL Especification. Retrieved March 07, 2013, from <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf>
- OASIS. (2010). XACML 3.0. Retrieved May 11, 2014, from https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml
- Olson, L. E., Gunter, C. a., & Madhusudan, P. (2008). A formal framework for reflective database access control policies. *Proceedings of the 15th ACM conference on Computer and communications security - CCS '08*, 289. doi:10.1145/1455770.1455808
- Olson, L., Gunter, C., Cook, W., & Winslett, M. (2009). Implementing reflective access control in SQL. *Data and Applications ...*. Retrieved from http://link.springer.com/chapter/10.1007/978-3-642-03007-9_2

- Oracle. (n.d.-a). The Java Language Environment. Retrieved April 17, 2014, from <http://www.oracle.com/technetwork/java/intro-141325.html>
- Oracle. (n.d.-b). Netbeans IDE. Retrieved April 20, 2014, from <https://netbeans.org/>
- Oracle. (n.d.-c). Java EE. Retrieved from <http://www.oracle.com/technetwork/java/javaee/overview/index.html>
- Oracle. (n.d.-d). RMI - Remote Object Invocation. Retrieved from <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html>
- Oracle. (n.d.-e). The Reflection API. Retrieved January 06, 2014, from <http://docs.oracle.com/javase/tutorial/reflect/>
- Oracle. (n.d.-f). Annotations. Retrieved March 03, 2014, from <http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html>
- Oracle. (n.d.-g). Getting Started with the Annotation Processing Tool. Retrieved March 04, 2014, from <http://docs.oracle.com/javase/7/docs/technotes/guides/apt/GettingStarted.html>
- Oracle. (1997a). JDBC Overview. Retrieved December 30, 2013, from <http://www.oracle.com/technetwork/java/overview-141217.html>
- Oracle. (1997b). JDBC Introduction. Retrieved March 03, 2014, from <http://docs.oracle.com/javase/tutorial/jdbc/overview/index.html>
- Oracle. (2006). The Java Persistence API. Retrieved March 03, 2014, from <http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html>
- Padma, J., & Silva, Y. (2009). Hippocratic PostgreSQL. *Data Engineering, 2009. ...*, (iii). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4812571
- Pereira, Ó., Aguiar, R., & Santos, M. (2011). CRUD-DOM: a model for bridging the gap between the object-oriented and the relational paradigms: an enhanced performance assessment based on a case study, *4*(1), 158–180. Retrieved from <http://ria.ua.pt/handle/10773/7959>
- Pereira, Ó., Aguiar, R., & Santos, M. (2012). ACADA: access control-driven architecture with dynamic adaptation. *SEKE'12 - 24th Intl. Conf. on Software Engineering and Knowledge Engineering*, 387–393. Retrieved from <http://repositorium.sdum.uminho.pt/handle/1822/19866>
- Pereira, Ó., Aguiar, R., & Santos, M. (2013). Runtime values driven by access control policies: statically enforced at the level of relational business tiers. *SEKE'13 - Intl. Conf. on Software Engineering and Knowledge Engineering*, 1–7. Retrieved from <http://repositorium.sdum.uminho.pt/handle/1822/25069>
- Pereira, O. M., Aguiar, R. L., & Santos, M. Y. (2012). ORCA: Architecture for Business Tier Components Driven by Dynamic Adaptation and Based on Call Level Interfaces. *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, (1), 183–191. doi:10.1109/SEAA.2012.22
- Pereira, Ó. M., Regateiro, D. D., & Aguiar, R. L. (2014a). Role-Based Access Control Mechanisms.

- Pereira, Ó. M., Regateiro, D. D., & Aguiar, R. L. (2014b). Extending RBAC Model to Control Sequences of CRUD Expressions.
- Reflection (C# and Visual Basic). (n.d.). Retrieved April 14, 2014, from <http://msdn.microsoft.com/en-us/library/ms173183.aspx>
- Rizvi, S., Mendelzon, A., Sudarshan, S., & Roy, P. (2004). Extending query rewriting techniques for fine-grained access control. *Proceedings of the 2004 ACM SIGMOD international conference on Management of data - SIGMOD '04*, 551. doi:10.1145/1007568.1007631
- Roichman, A., & Gudes, E. (2007). Fine-grained access control to web databases. *Proceedings of the 12th ACM symposium on Access control models and technologies - SACMAT '07*, 31. doi:10.1145/1266840.1266846
- Samarati, P., & Vimercati, S. de. (2001). Access control: Policies, models, and mechanisms. *Foundations of Security Analysis and* Retrieved from http://link.springer.com/chapter/10.1007/3-540-45608-2_3
- Sandhu, R., & Coynek, E. (1996). Role-Based Access Control Models. *IEEE ...*, 29(2), 38–47. Retrieved from <http://modelibra.googlecode.com/svn/trunk/Modelibra/doc/security/sandhu96.pdf>
- Schneier, B. (1998). The Blowfish Encryption Algorithm. *Dr. Dobbs Journal*, 23, 38–40. Retrieved from http://www.ddj.com/ddj/1998/1998_12/../../../../ftp/1998/1998_12/twofish.zip
- Schneier, B., Kelsey, J., Whiting, D., Wagner, D., & Hall, C. (1998). Twofish : A 128-Bit Block Cipher. *Current*, 21, 1–27. doi:10.1.1.35.1273
- Shneiderman, B. (1984). Response time and display rate in human performance with computers. *ACM Computing Surveys (CSUR)*, 16(3). Retrieved from <http://dl.acm.org/citation.cfm?id=2517>
- Sumathi, S., & Esakkirajan, S. (2007). *Fundamentals of relational database management systems*. Retrieved from <http://books.google.com/books?hl=en&lr=&id=RjnNA0GW0wsC&oi=fnd&pg=PA1&dq=Fundamentals+of+Relational+Database+Management+Systems&ots=nlbyTWX5AY&sig=Qj7jDSWbGD9ogbGXRW3MkDZw4xU>
- The Legion of the Bouncy Castle. (n.d.). Retrieved April 25, 2014, from www.bouncycastle.org
- Vimercati, S. di, Foresti, S., & Samarati, P. (2007). Authorization and access control. *Security, Privacy, and Trust in* Retrieved from http://link.springer.com/chapter/10.1007/978-3-540-69861-6_4
- W3C. (n.d.). Web Services Choreography Description Language. Retrieved March 07, 2014, from <http://www.w3.org/TR/ws-cdl-10/>
- Wallach, D. S., Appel, A. W., & Felten, E. W. (2000). SAFKASI: a security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(212), 341–378. doi:10.1145/363516.363520

- Wei, Q., Crampton, J., Beznosov, K., & Ripeanu, M. (2011). Authorization recycling in hierarchical RBAC systems. *ACM Transactions on Information and System Security*, 14(1), 1–29. doi:10.1145/1952982.1952985
- Wu, T. (1998). The Secure Remote Password Protocol. In *Proceedings of the Symposium on Network and Distributed Systems Security NDSS 98* (pp. 97–111). Retrieved from <ftp://srp.stanford.edu/pub/srp/srp.ps>,
- XML Process Definition Language. (n.d.). Retrieved March 07, 2014, from <http://www.xpdl.org/>
- YAWL. (n.d.). Retrieved March 07, 2014, from <http://yawlfoundation.org/>
- Zarnett, J., Tripunitara, M., & Lam, P. (2010). 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*, 79. doi:10.1145/1809842.1809858
- Zhang, G., & Parashar, M. (2003). Dynamic context-aware access control for grid applications. *Grid Computing*, 2003. *Proceedings*. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1261704
- Zhu, Y., Hu, H., Ahn, G.-J., Yu, M., & Zhao, H. (2012). JIF: Java + information flow. Retrieved from <http://www.cs.cornell.edu/jif/>

Appendix A – SSL/TLS and authentication implementation details

In this section we will present the implementation details regarding the standard authentication protocols and the integration of the SSL/TLS protocol into the authentication process.

This section is divided as follows: Section A.1 presents the implementation details of the authentication method using the hash-based password authentication protocol, section A.2 presents the implementation details of the challenge-response authentication mechanism, section A.3 presents our implementation of a TFA method using the CC and section A.4 presents the integration of the SSL/TLS protocol into the authentication phase.

A.1 Hash-based password authentication protocol

This section will present the implementation details regarding the hash-based password authentication protocol.

The implemented protocol, shown in Table 18, starts by sending a *GetSalt* message (1), indicating its username. The server, having the salt of the user available, returns the salt to the client (2). Then the client indicates that it wants to authenticate using the *Plain* method by sending an *AuthPlain* message (3), with the application name, the user's username and hashed password, the IP address and the listening port to receive notifications about policy changes. The server then sends the message *Authenticating* if the authentication method is allowed or *Not_Allowed* if it is not (4), thus terminating the handshake. If the authentication method is supported, then the server compares the received hashed password (s') with the hashed password that it knows (S), sending an *OK* message if they match and an *NOK* message otherwise (5).

Table 18. Hash-based Password Authentication Protocol messages.

#	Client	Network	Server
1	username, s	→ GetSalt username	
2	username, s, salt	← salt	username, $S = \text{Hash}(s + \text{salt})$, salt
3	username, s, salt, $s' = \text{Hash}(s + \text{salt})$	→ AuthPlain appName username s' IP port	username, s' , S , salt
4	username, s, s' , salt	← Authenticating / Not_Allowed	username, s' , S , salt
5	username, s, s' , salt	← OK/NOK	username, s' , S , salt

A.2 Challenge-response authentication mechanism

This section will present the implementation details regarding the challenge-response authentication mechanism protocol.

The protocol implemented to use this mechanism is shown in Table 19 and starts by sending a *GetSalt* message indicating the user's username (1). The server then replies with the salt value used to hash the password (2). The client then sends an *AuthChallenge* message with its username and challenge to the server (3). If the server allows this authentication mechanism to be used then it

replies with an *Authenticating* message or a *Not_Allowed* message otherwise (4). If it is allowed then the server sends (5) its challenge (s_chal) and the response for the client's challenge (s_resp). The client then calculates the expected response, if it matches the received response then the client sends a *AuthChallengeResponse* message with its response for the server's challenge (c_resp), if not, a *NOK* is sent to terminate the handshake (6). Finally, the server calculates the expected response from the client and sends an *OK* if it matches or a *NOK* if it does not match (7). Since the responses only match if each endpoint knows the shared secret (the hash of the password) then this mechanism authenticates both the client and the server.

A requirement is imposed on the generated challenges: the client challenge must be even (i.e. the last bit has to be 0) and the server challenge odd (i.e. the last bit has to be 1). This requirement prevents the execution of reflection attacks using this authentication mechanism.

Table 19. CRAM based authentication messages.

#	Client	Network	Server
1	username, s	→ GetSalt username	
2	username, s, salt	← salt	username, S = Hash(s + salt), salt
3	username, s, salt, $s' = \text{Hash}(s + \text{salt})$, $c_chal = \text{random}()$	→ AuthChallenge username c_chal	username, S, salt, c_chal
4	username, s, s' , salt, challenge	← Authenticating / Not_Allowed	username, S, salt, c_chal
5	username, s, s' , salt, c_chal, s_chal , s_resp , $c_resp = \text{hash}(c_chal + s_chal + s')$	← s_chal ← s_resp	username, S, salt, c_chal, $s_chal = \text{random}()$, $s_resp = \text{hash}(s_chal + c_chal + S)$
6	username, s, s' , salt, c_chal, s_chal , s_resp , c_resp	→ AuthChallengeResponse c_resp / NOK	username, S, salt, c_chal, c_resp , s_resp , s_chal
7	username, s, s' , salt	← OK/NOK	username, s' , S, salt

A.3 Two factor authentication

This section will present the implementation details regarding the TFA authentication protocol and mechanism.

There are five different messages used in this context: a "TFA" message sent by the server to indicate that a TFA is required, a "TFACert <certificate>" message sent by the client to provide the server with its public key certificate, a "<challenge>" sent by the server for the client to sign in order to be authenticated, a "AUTH_DENIED" message sent by the server if the public key certificate BI information does not match the stored BI for that user and a "TFASign <signature>" message sent by the client to send the signature to the challenge.

Table 20 shows how these messages are used to perform a TFA (data related to the first authentication process is omitted for simplification). When a client is authenticating, there are several messages that are exchanged between the client and the server (1). Every authentication method mentioned ends with the server sending an *OK* or a *NOK* message to the client (6). Right before this message the server, if configured to do so, it can send a *TFA* message requesting a TFA (2). The client

must read the authentication certificate from the user's CC and send it in a *TFACert* message to the server, whom then proceeds to validate the certificate (3).

Each certificate in every CC is signed by a chain of government certificates, whose public key certificates are publicly available. The server, when it receives the user's authentication certificate, checks its signature using the public key stored in the government's public key certificate that has supposedly signed it. For this purpose the certificate path validator (*CertPathValidator*) is used. Using the *CertPathValidator* we can set our trust anchor, to which we set the government's certificate that the server possesses, and build the chain of certificates to validate the user's certificate that we received. Finally, since the government has an OCSP endpoint publicly available which we can use, we also activate the OCSP protocol to check the revocation status. After validating the user's certificate we then extract the BI from it can compare it with the BI stored in the database. If they match then the server sends a challenge to the client (4), who signs it and sends the signature back (5), otherwise the server sends an *AUTH_DENIED* message and the authentication fails. Because the server has the public key certificate validated, it can verify the received signature. If the signature matches, the TFA succeeds and the server then sends the *OK* message. Otherwise a *NOK* message is sent instead (6).

Table 20. Two factor authentication protocol.

#	Client	Network	Server
1		↔ Authentication mechanism	
2	cert	← TFA	
3	cert	→ TFACert <certificate>	BI, cert, certCA, challenge
4	cert	← <challenge> / AUTH_DENIED	BI, cert, certCA, challenge
5	cert, challenge, signature	→ TFASign <signature>	BI, cert, certCA, challenge
6	cert, challenge, signature	← OK / NOK	BI, cert, certCA, challenge, signature

A.4 SSL/TLS with certificates

This section will present the implementation details regarding the protocol to activate the SSL/TLS protocol during the authentication phase.

This method, shown in Table 21, starts like the others, by using an unsecure and a non-encrypted channel. First the client sends a request to upgrade the channel using SSL/TLS by sending a *UP_SSL* message (1). The server then replies with *Authenticating* if it allows the upgrade or *Not_Allowed* if not (2). Then the server sends the port where it will be listening for the new connection and terminates (3). The client then connects to the new port and starts the SSL/TLS handshake process, which is provided by the Java language. If the loaded public key certificate contains the public key that matches the private key used by the server, then the server is authenticated and a session key agreed that will encrypt further communications. After that we still require to authenticate the client and to make sure that the server is not an impostor that was able to replace the public key certificate on the client side, so we start the challenge-response mechanism proposed in section 4.4.1.2.

Table 21. SSL/TLS with certificates upgrading protocol.

#	Client	Network	Server
1	username, s	→ UP_SSL	
2	username, s	← Authenticating / Not_Allowed	
3	username, s	← port	
4	username, s	↔ Challenge-Response mechanism	

Appendix B – CoherentPaaS project and its applicability

In this appendix we will introduce CoherentPaaS(“CoherentPaaS,” n.d.), a European project about a platform that aims to provide a coherent view of a cloud environment by providing an abstraction layer for several different data stores, in which we are involved in.

This appendix is divided as follows: Section B.1 will provide an overview of the CoherentPaaS protect and section B.2 will discuss the potential applicability of it as a data store abstraction layer for S-DRACA and the work done on top of it.

B.1 Overview

In this section we will present an overview of the CoherentPaaS project and the goals it aims to achieve.

CoherentPaaS is a European project with several different development branches. It has a common query engine, which will allow applications to access data from different data stores and cloud services in a unified and integrated manner through a common query engine. It also has a holistic transactional API to provide coherence between different data stores and orchestrates transactions transparently to the applications. This holistic transactional component will provide these transactional functionalities in an ultra-scalable manner while retaining the ACID (i.e. atomicity, consistency, isolation and durability) properties. Because this project will provide a rich PaaS with different data stores, which are optimized for particular tasks, data and workloads, the need to copy and translate big quantities of data from one data store to another is removed.

Another objective of this project is to abstract not only relational data stores, but non-relational data stores as well. It will also provide a complex event processing (CEP) engine which will be integrated into the common query engine, to consume data from the data stores as events and store the results back into a data store. The main challenges for this project is to enable the data stores to be supported (both relational, the non-relational and even the CEP engine) to provide some sort of transactional functionality and the design of the common query language, which will have to abstract very different data stores in terms of syntax and provided functionalities.

To test the correctness and performance of the components developed in the project, we will have several use cases which will use the CoherentPaaS for different purposes and with different needs.

B.2 Applicability

In this section we will discuss the applicability of the CoherentPaaS solution in the context of the work presented in this dissertation.

As we have discussed before in section 5.3.2, our work provides a way to design different Business Schema generators. Normally, we would require a different generator for different data

stores, unless the syntax of the querying language, the provided functionalities and the supported operations by the driver (e.g. JDBC) are the same. Furthermore, creating a generator for a non-relational data store is not always easy and requires some functionalities that are not initially supported to be implemented in the Business Schemas.

CoherentPaaS has the potential to make it trivial to apply our work to different data stores, by developing a Business Schema generator for the CoherentPaaS' own common query engine instead of each particular data store. However, the result of a select statement on the common query engine might include results from several data stores, including the results of non-relational data stores that were transformed into a table format, which we are not able to trace back (i.e. which data store provided which column data), so the insertion, modification and deletion of data through the indirect access mode might be a problem.