Charles University in Prague

Faculty of Mathematics and Physics

# MASTER THESIS

Ondřej Štumpf

## Security and Trust in the DEECo Component Model

Department of Distributed and Dependable Systems

Supervisor of the master thesis: doc. RNDr. Tomáš Bureš, Ph.D.

Study programme: Informatics

Specialization: Software Systems

Prague 2015

First and foremost I would like to thank my supervisor, doc. RNDr. Tomáš Bureš, Ph.D., for his patience, goodwill and numerous pieces of advice. The same amount of gratitude belongs to my family and girlfriend, whose huge support enabled me to create this work in the first place. Also, I would like to thank Mgr. Michal Kit and Mgr. Vladimír Matěna for their technical support.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In........... date........... signature

Název práce: Bezpečnost a důvěra v komponentovém modelu DEECo

Autor: Ondřej Štumpf

Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: doc. RNDr. Tomáš Bureš, Ph.D.

Abstrakt: DEECo je příkladem Cyber-Physical Systému (CPS), který se může skládat z potenciálně velkého množství komponent schopných sdílet mezi sebou data. Dosud nebyl jednotlivým komponentám nijak omezen přístup k datům, což jim umožňovalo zneužít citlivé informace vlastněné jinými komponentami. Cílem této práce je analyzovat bezpečnostní hrozby existující v distribuovaných prostředích podobných DEECo a navrhnout bezpečnostní řešení, které by zajišťovalo jak fyzickou bezpečnost dat, tak i řízení přístupu k nim. Zatímco však důvěrnost informací může být důležitá pro některé aplikace, jejich integrita je klíčová téměř pro všechny. V této práci je proto dále navržen model řešící důvěru mezi jednotlivými komponentami, který zabraňuje použití chybných či podvržených dat. Oba návrhy jsou realizovány v systému jDEECo, implementaci DEECo v jazyce Java.

Klíčová slova: bezpečnost, důvěra, DEECo, komponenta

Title: Security and Trust in the DEECo Component Model

Author: Ondřej Štumpf

Department / Institute: Department of Distributed and Dependable Systems

Supervisor of the master thesis: doc. RNDr. Tomáš Bureš, Ph.D.

Abstract: DEECo represents an example of a Cyber-Physical System (CPS) consisting of potentially vast number of components able to share data with each other. So far, access to data was not restricted, thus enabling components to exploit sensitive data owned by other components. The goal of this work is to analyze security threats in distributed environments such as DEECo and propose a security solution which would provide both physical security of component data and introduce an access control mechanism. However, while confidentiality may be critical to certain applications, data integrity is crucial to almost every one. This work therefore also proposes a trust model, which prevents components operating with defective or malicious data. Both proposed models are realized in jDEECo, a Java implementation of DEECo.

Keywords: security, trust, DEECo, component

# Contents

1 Introduction ... 1
1.1 Problem Description ... 1
1.2 Goals ... 2
1.3 Thesis Organization ... 2
2 Background: DEECo ... 3
2.1 Running Example ... 3
2.2 Key Concepts ... 3
2.3 Components ... 3
2.4 Ensembles ... 5
2.5 Knowledge ... 6
3 Problem Analysis: Security and Trust in DEECo ... 8
3.1 Security Threats ... 8
3.1.1 Unauthorized Access ... 8
3.1.2 Data Leakage ... 8
3.1.3 Data Manipulation ... 9
3.1.4 Fake Messages ... 9
3.1.5 Replay Attacks ... 9
3.1.6 Covert Channels ... 9
3.1.7 Runtime Corruption ... 10
3.2 Trust Concerns ... 10
3.2.1 Integrity Assurance ... 11
4 Background: Security Models ... 12
4.1 Genealogy ... 12
4.2 Single-level Models ... 13
4.2.1 Reference Monitor ... 13
4.3 Mandatory Access Control (MAC) ... 14
4.3.1 Bell-LaPadula ... 14
4.3.2 Biba ... 15
4.4 Discretionary Access Control (DAC) ... 16
4.5 Role-Based Access Control (RBAC) ... 17
4.6 Clark-Wilson ... 19
4.7 Brewer-Nash (Chinese Wall) ... 21
4.8 Distributed Security Models ... 23
4.8.1 Capability-Based Access Control ... 23
4.8.2 Credential-Based Access Control ... 24

5 Background: Trust Management ... 26
5.1 PolicyMaker and KeyNote ... 26
5.2 SULTAN ... 28
6 Background: Cryptography ... 29
6.1 Symmetric Cryptography ... 29
6.2 Asymmetric Cryptography ... 30
6.3 Digital Signatures ... 31
7 Solution Strategy ... 32
8 Realization ... 33
8.1 Assumptions ... 33
8.2 Principles ... 33
8.3 Security Architecture ... 34
8.3.1 Security Policy Specification ... 34
8.3.2 Encryption and Signing ... 37
8.3.3 Access Control ... 42
8.3.4 Component Clearance Verification ... 51
8.3.5 Data Leakage Prevention ... 54
8.4 Trust Architecture ... 57
8.4.1 Concept ... 57
8.4.2 Obtaining the Rating ... 58
8.4.3 Creating the Rating ... 59
8.4.4 Ratings Distribution ... 61
9 Evaluation ... 62
10 Discussion ... 67
11 Related Work ... 69
11.1 dRBAC ... 69
11.2 RT Framework ... 69
11.3 IoT Security ... 70
12 Conclusion and Future Work ... 72
13 Bibliography ... 73
14 List of Figures ... 75
15 List of Abbreviations ... 76
16 Attachments ... 77
17 Appendix – Build Instructions ... 78

# 1 Introduction

The concept of security has been a vital issue ever since computer systems were introduced. In the early days, each computer system represented a standalone independent unit, where security usually meant concealing data of one user from the others. Purely centralized solutions developed to satisfy such requirements soon became obsolete, as the computer systems became interconnected and necessity to enforce security and data integrity across multiple physical devices arose.

Cyber-Physical Systems (CPS) form the next evolutionary step with its own set of security issues. CPS consist of autonomous entities which have the ability to communicate with one another. Each entity is often provided access to mission critical data, collected from various sensors or obtained from other entities. Preventing data compromise and ensuring data integrity is therefore a crucial part of CPS development. Since no central authority or management is present, security must be achieved by collaboration of the entities themselves.

In this thesis we focus on one particular example of CPS – DEECo (Distributed Emergent Ensembles of Components). We analyze security vulnerabilities that DEECo shares with other CPS and identify issues that are DEECo-specific. A solution providing both physical security and data access control is then proposed. We also examine perils to data integrity and devise a system enabling components to modify their behavior according to the quality of information they receive.

Suggested mechanisms are then implemented in the current version of jDEECo, the Java implementation of DEECo.

## 1.1 Problem Description

Since no central authority exists in the DEECo component model, security and data integrity can only be achieved by participating components and the DEECo runtime. This involves [1]:

- Storage Security – each component's data must prevented from unauthorized access and tampering.
- Communication Security – data transmitted between components must be resilient to traffic monitoring and data forging attacks.
- Access Control – only authorized components may gain access to protected data.

- Data Leakage Mitigation – manipulation with data with certain level of security cannot lead to its declassification.
- Data Integrity Enforcement – data violating integrity constraints must not jeopardize other data in the system. The very notion of what "integrity constraints" actually means must be left for a component using the data to define.
- Open-endedness and Dynamism – the system is not strictly defined, new components with new security and trust requirements may be deployed at any time. A component does not know the other components it may communicate with at deploy time – the security specification cannot be based on any hard-coded relations.

## 1.2 Goals

The goal of this thesis is to analyze security threats and data integrity risks in DEECo and propose a suitable mechanism for ensuring data security and integrity. We will examine well-known security and trust models and discuss their suitability for utilization in DEECo. The proposed solution will be implemented in jDEECo and verified on real-world case study.

## 1.3 Thesis Organization

The rest of the thesis is organized as follows. In chapter 2, we introduce DEECo and describe its core features and principles, necessary for determining security risks. In chapter 3, we analyze security and trust issues in DEECo and distributed environments in general. Chapters 4 and 5 are dedicated to the description of well-known security and trust models, respectively. In chapter 6 we provide the very basics of cryptography, used later in description of the physical security of data. This theoretical background knowledge is then utilized in chapter 7, which briefly outlines the proposed solution. In chapter 8, we describe the proposed solution in detail, providing examples and eventually evaluating the solution. The last chapters are dedicated to discussing advantages and disadvantages of the proposed solution (chapter 10), mentioning several security models similar to ours (chapter 11) and summarizing the thesis as a whole.

# 2 Background: DEECo

In this chapter we describe the component model and general concepts of DEECo, which we later utilize to propose a solution for security and trust concerns. A running example depicted in the following section is used throughout the whole thesis to demonstrate various aspects of DEECo and its security challenges. It utilizes jDEECo, the Java implementation of DEECo which is also used for realization of security solution, described in chapter 8.

## 2.1 Running Example

We assume the following setup: there are two kinds of police, the State Police (SP) and the Municipal Police (MP). For our intents and purposes, the SP is responsible for chasing criminals, while the MP imposes speeding tickets and tickets for bad parking. Obviously there are also ordinary vehicles on the roads, which are monitored by both kinds of police. While the SP has jurisdiction everywhere, the MP is always bound to a single city.

In the following sections, we will use this setup to demonstrate various features of DEECo.

## 2.2 Key Concepts

DEECo represents an example of an Ensemble-Based Component System (EBCS). As such it consists of autonomic entities (called components) that are dynamically composed into cooperative groups (called ensembles). The system architecture is then represented as components bound through ensembles. This architecture emerges at runtime, i.e. ensembles are dynamically created based on component data.

An essential feature of DEECo is its communication paradigm – no components can communicate directly, the only way of exchanging data is through ensembles. The data exchange as well as all the other management services are handled by the DEECo runtime. [2]

## 2.3 Components

A component is an independent and autonomous unit, which is deployed and maintained by the DEECo runtime. Each component maintains its view of the system, i.e. subset of globally available data, called *knowledge*. It may not communicate

directly with any other component, the only external source of knowledge is the exchange operation within an ensemble (see 2.4).

A component may define one or more *processes*. A process can both access and modify component's knowledge, which it uses to determine component's behavior. A process cannot access knowledge of other components. It can either be run periodically or it can be triggered by knowledge change.

Knowledge forms a flat space, where no pointers or references are allowed. Especially, a component itself cannot be a part of knowledge of another component.

The example below depicts a component that corresponds to the State Police vehicle from the running example. The component itself owns a list of wanted criminals and through the *PoliceRadar* ensemble (see 2.4) it obtains the identifier of a nearby vehicle. Periodically scheduled process than checks if such vehicle belongs to a wanted criminal and if so, instructs the State Police component to begin pursuit.

```
@Component
public class StatePolice {

	public String pursuedCriminal;
	public String vehicleNearbyDriver;
	public Set<String> wantedCriminals;

	public StatePolice(Set<String> wantedCriminals) {
		// create the police with given list of wanted criminals
		this.wantedCriminals = wantedCriminals;
	}

	@Process
	@PeriodicScheduling(period = 1000)
	public static void startPursuitIfCriminalNearby(
		@In("wantedCriminals") Set<String> wantedCriminals,
		@In("vehicleNearbyDriver") String vehicleNearbyDriver,
		@InOut("pursuedCriminal") ParamHolder<String> pursuedCriminal
	) {
		// if the police does not chase anyone at the moment
		// and the nearby vehicle is driven by a wanted criminal
		if (pursuedCriminal.value == null &&
		wantedCriminals.contains(vehicleNearbyDriver)) {
			// start pursuit
			pursuedCriminal.value = vehicleNearbyDriver;
		}
	}
}
```

*Figure 1: Example of the State Police component*

## 2.4 Ensembles

An ensemble represents a relation between two components, one of them acting as *coordinator*, the other as *member*. An ensemble is defined by a predicate taking components' knowledge as an input and determining whether such two components can form the given ensemble. This predicate is called a *membership condition*. A membership condition may be asymmetrical, since it referrers explicitly to coordinator's and member's knowledge (which may be completely different).

An ensemble also defines a *knowledge exchange* operation. This operation is allowed to modify both member's and coordinator's knowledge, possibly transferring data from one to the other. A knowledge exchange can only be performed right after the membership condition (from the corresponding ensemble of course) was invoked and satisfied.

If certain part of knowledge required by a membership condition is missing from the target component, the membership condition is always evaluated to false. Similarly, a knowledge exchange cannot be performed if any transferred knowledge is missing.

In our running example, an ensemble could be formed from vehicles within certain distance (for instance within a range of a police radar), allowing the police to gather information about vehicle's owner, speed etc. Alternatively, vehicles sharing the street could form an ensemble to exchange information about available parking space.

```
@Ensemble
@PeriodicScheduling(period = 5000)
public class PoliceRadar {

	@Membership
	public static boolean membership(
		@In("member.position") Coord memberPosition,
		@In("coord.position") Coord coordPosition) {
		// compute distance and return true if within range
	}

	@KnowledgeExchange
	public static void exchange(
		@In("member.ownerName") String ownerName,
		@Out("coord.vehicleNearbyDriver")
			ParamHolder<String> vehicleNearbyDriver) {
		// use "member." to access member's knowledge
		// use "coord." to access coordinator's knowledge
		vehicleNearbyDriver.value = ownerName;
	}
}
```

*Figure 2: Example of ensemble - police radar*

## 2.5 Knowledge

Knowledge forms a flat global space, each component owns a copy of subset of this space. To address a particular part of the knowledge, a concept of *knowledge paths* is used. A knowledge path can either be absolute, i.e. refer directly to data, or it may contain evaluable expressions – nested knowledge paths, whose data then form the resulting absolute path. For example, let us consider the following component definition:

```
@Component
public class OrdinaryVehicle {
	public String id;
	public String interestingProperty;
	public List<String> passengerNames;
	public Map<String, Object> properties;
	public Coordinates position;

	@Local
	public String driverId;
}
public class Coordinates {
	public int x;
	public int y;
}
```

*Figure 3: Example component for demonstrating knowledge path variants*

The following knowledge paths are valid examples:

- *id* – accesses the *id* field
- *position.x* – accesses the *x* field of the *Coordinates* object
- *passengerNames.0* – accesses the first passenger name
- *properties.manufacturer* – gets the value the key *manufacturer* from the *properties* map
- *properties.[interestingProperty]* – evaluates the nested path in the brackets and then the surrounding path. If the value of *interestingProperty* was "color", the resulting knowledge path would be *properties.color*.

Any knowledge field may be decorated with the *@Local* annotation. This implies that this particular field will be available to the component processes the same way as any other, but it will not be distributed to other components in the system.

We specifically stress the knowledge path expressibility, because it has a serious impact on security. When accessing data through a knowledge path containing nested expressions, security level of all partial knowledge paths must be taken into account.

As a result of that, information about security protection of the knowledge must be safely distributed across components so that when a component attempts to access data through knowledge exchange, the runtime can make sure the access is legitimate.

# 3 Problem Analysis: Security and Trust in DEECo

In this section we analyze threats to data security and integrity in DEECo. Some perils stem from the fact that DEECo is a distributed system, some originate from the dynamism of the DEECo architecture. For each listed risk we determine the requirements for security system so that it can guarantee immunity of DEECo to that risk.

## 3.1 Security Threats

### 3.1.1 Unauthorized Access

Data confidentiality and is the obvious concern when proposing a security system. In the DEECo context, this implies:

- No component can access other component's knowledge without a proper security clearance.
- No other system process sharing the hardware with the DEECo runtime can access any of the DEECo data.
- Secured data transferred between DEECo entities must not be read nor modified without the runtime noticing (man-in-the-middle attacks resistance).

### 3.1.2 Data Leakage

The term *data leakage* commonly refers to a situation where classified information loses its protection during transmission and becomes accessible with lesser security privileges than originally required [3].

To prevent this, the proposed security system for DEECo must ensure that whenever a component is entrusted with knowledge, the knowledge will not lose its security status and will not eventually become accessible for any other component without proper access rights. Obviously, the knowledge exchange operation is critical from this point of view. However, we must also consider the effect of component processes, since they can modify the knowledge and therefore an "evil" component could safely obtain data through knowledge exchange and then make it available for anyone else by copying it to an unsecured part of knowledge.

### 3.1.3 Data Manipulation

It is not only necessary to prevent the data from unauthorized reading, it must also be impossible for an intruder to modify the data without the runtime noticing. This is especially important during communication, where some form of read access is virtually inevitable. Particularly in DEECo, modification of any of these properties of a message must be prevented:

- Source component identifier
- The data version
- The knowledge data itself along with their knowledge paths
- The security metadata of the knowledge data

### 3.1.4 Fake Messages

It must be impossible to forge fake messages and send them to the DEECo runtime. Also, a component must be prevented from impersonating another component.

### 3.1.5 Replay Attacks

A replay attack consists of capturing and storing a transmitted message and using it again after some time. A popular example of viciousness of such attack is a payment order – if an attacker captured a message containing such information and used it again, the payment order would eventually be performed twice.

However the way DEECo transmits data is idempotent (i.e. its repetitive execution would not change the result more than once). Since we are guaranteed data version safety (see 3.1.3), the replay attacks need not concern us.

### 3.1.6 Covert Channels

The term *covert channel* commonly refers to a situation where a subject can infer partial information about an object it cannot legally access. This could be for example the very existence of such object, the value it is *not* equal to, the range it falls in etc.

For example if a police car is seen with its sirens on, we can deduce that it is chasing a criminal, even though we do not know the name of the criminal.

This problem is actually closely related to the data leakage problem discussed in 3.1.2. If the component (police car) had a process which would decide whether the car's sirens should be on, an input of this process would be the name of the pursued

criminal (secured knowledge) and the output would be the state of the sirens (inherently unsecured). Then such process actually violates the classification of the secured knowledge.

However, the only solution for this problem would be to take the police car's sirens off or make the state of the sirens a secured knowledge – both of which is rather undesirable. The devised DEECo security system must therefore enable components to partially control the data leakage verification process to enable "safe" data leakage the component is aware of.

### 3.1.7 Runtime Corruption

We must consider a situation where the whole DEECo runtime was hacked and cannot be relied upon. The term often discussed in these circumstances is TCB – Trusted Computing Base, which generally comprises the hardware and software critical to security. OMG (Object Management Group) comments this:

*"The TCB should be kept to a minimum, but is likely to contain operating system(s), communications software (though note that integrity and confidentiality of data in transit is often above this layer), (…), security services and other object services called by any of the above during a security relevant operation."* [4]

The TCB can eventually become quite large and obviously we want to avoid this. However, we have no choice but to trust the local runtime, local operating system, the device hardware etc. Nevertheless, this does not apply to the DEECo input (i.e. definition of components and ensembles), which are inherently untrusted. In the proposed security system, whenever a component utilizes any security feature of DEECo, identity of the component author must be verified before deploying such component into the runtime (for example via a digital signature).

## 3.2 Trust Concerns

So far we considered only two absolute states of knowledge – the safe state, where confidential knowledge is secured and accessed only by authorized components, and the risk state, where there is some kind of security breach (any of those discussed in 3.1). These two states can be objectively judged and distinguished.

In most systems however, this taxonomy is not sufficient. Even if all security requirements are fulfilled, there may be a reason for a component not to use the data it received, for example if it led to a violation of component's inner integrity constraints.

Hence, a trust management system needs to be devised to enable components share information about trustworthiness of the knowledge.

Trustworthiness, unlike security, is not absolute, but graded. Moreover, it is relative – in the real world, we trust someone with something, but that does not imply that everyone else trusts them the same. The trust management system we propose needs to reflect all these properties.

### 3.2.1 Integrity Assurance

Since DEECo is a representative of CPS, we can expect that at least part of knowledge will consist of data collected from device sensors. These sensors can be inherently faulty and provide malformed data. It is necessary to provide such a trust model that would enable components reason about quality and trustworthiness of knowledge they own.

# 4 Background: Security Models

First, let us define the term *security policy*: “*A security policy is nothing more than a well-written strategy on protecting and maintaining availability to your network and its resources.*” [5] Building on that, we consider *security model* to represent a formal description of the security policy. Our endeavor to make DEECo secure then consists of identifying its assets, discovering potential threats and proposing a security policy, which would satisfy our idea of the system being “secure”.

In this section, we will describe well-known security models and discuss their suitability for usage in distributed systems and eventually in DEECo.

## 4.1 Genealogy

The need for serious data confidentiality in information systems arose from the military sector in the 1970s. The early research of this topic therefore primarily focused on access control and information leakage detection and prevention. In this era, Bell and LaPadula introduced their multilevel security model to match the needs of government and military applications [6]. Also, the legendary “Orange Book”, issued by the Unites States Government Department of Defense (DoD) as Trusted Computer System Evaluation Criteria (TCSEC), was created to evaluate and classify computer systems security.

While data confidentiality is essential for military sector, in commercial sector, data integrity is considered at least equally important. The Biba model [7] is a prime example of shifting from data security to data integrity – the Biba model forms a dual model to the previously mentioned Bell-LaPadula model, focusing on data trustworthiness and completely omitting confidentiality. Also, models to deal with even more business-specific demands were designed, for example the Chinese Wall model (see 4.7), which prevents the insider knowledge to cause conflict of interests between business parties.

Reflecting the business sector demands, role-based access control models were introduced in the 1990s to better match with organizational structure of corporations. A role in such model corresponds with a position in a company hierarchy and is a basic unit when setting up security policies. [8]

As networked systems began to be used, necessity for decentralized security models lead to the introduction of capability-based systems and eventually

credential-based access control, which includes for example the X.509 certificates being used today.

## 4.2 Single-level Models

Single-level models recognize only one level of security classification, i.e. an object either is or is not accessible by the given subject. In such simple setup, no roles or hierarchy is considered.

### 4.2.1 Reference Monitor

Any subject demanding access to any potentially sensitive object must first invoke a *monitor*, to which it passes the description of what object it wants to access and what operations it wants to perform. The monitor evaluates the request and responds with a simple yes/no, possibly auditing the request. [9]

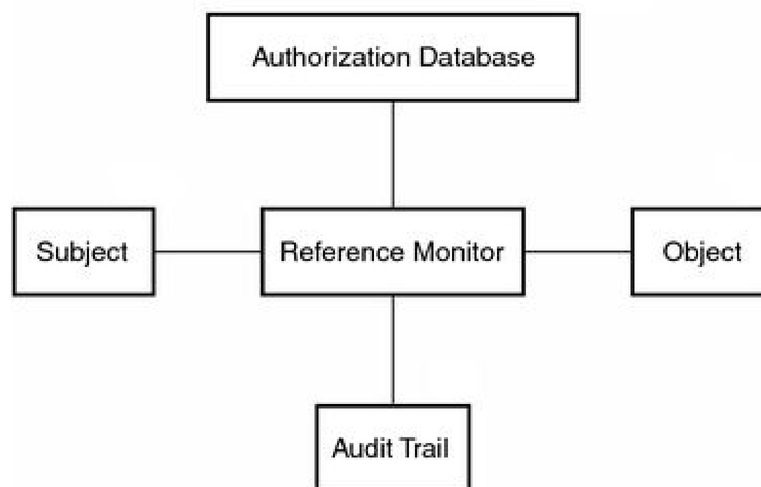


*Figure 4: The Reference Monitor model schema*

The properties the reference monitor must have in order to guarantee security can be described by the NEAT acronym:

- Non-bypassable – there is no way for a subject to access an object without having consulted the reference monitor first.
- Evaluable – the model must be testable and verifiable.
- Always invoked – the reference monitor is invoked on every access to an object.
- Tamper-proof – it must not be possible for an intruder to modify reference monitor configuration, code, or data.

Obviously, the reference monitor must be available for all subjects in the system at any given time. Also, the reference monitor is invoked on every access to any object (non-bypassable and always invoked) and therefore it may represent a serious bottleneck and single point of failure. For this reasons, this model is not suitable for any usage in a distributed environment. [10]

## 4.3 Mandatory Access Control (MAC)

Multi-level security models were originally developed with military applications in mind. In such organizations, every piece of information is assigned exactly one level of confidentiality, for example *unclassified*, *confidential*, *secret* or *top secret*. Similarly, military personnel are assigned a clearance level, which may depend on rank, unit etc. The system then enforces *the principle of least privilege* – a subject gets only such clearance that is necessary for their work and they may access only objects with security level equal to or less than their clearance.

In the following sections, we describe two dual security models, both building on the premise of data classification. While the Bell-LaPadula model was designed to provide data confidentiality, the Biba model equally focuses on data integrity.

### 4.3.1 Bell-LaPadula

In the Bell-LaPadula model [6], the system is described as a state machine. Initially, the system is in “secure state” (whatever that means for the particular application). It is then proven that formalized transition functions always end up with system being again in secure state, thus preserving the security policy.

More formally, the model consists of the following elements:

- The ordered set of *classifications* (confidential, secret, top secret etc.)
- The set of *compartments*, describing logical units within the system (Secret Service, government, military research etc.).
- *Objects*, which are assigned a set of compartments and a classification. The tuple (compartments, classification) then forms the *security level* of the object. For example, the information about eavesdropped phone calls could have the security level ({Secret Service}, confidential).
- *Subjects*, whose security level is again the tuple (compartments, classification).

The model then defines partial ordering on security levels:

$$\forall \text{a}, \text{b} \in \text{security levels}, \ \text{a} \leq \text{b} \ \stackrel{\text{def}}{=} (a.classification \leq b.classification) \\ \& \ (a.compartments \subseteq b.compartments)$$

Finally, the following properties must be hold to consider a transition function secure:

- *The Simple Security Property* – for a security level $SL_s$ of a subject *reading* an object with a security level $SL_o$, it must hold that $SL_o \leq SL_s$. This is also known as the "no read-up" property.
- *The *-property* (read as the "star property") – for a security level $SL_s$ of a subject *writing* to an object with a security level $SL_o$, it must hold that $SL_o \geq SL_s$. This is also known as the "no write-down" property.

The purpose of the Simple Security Property is obvious – no subject may read an object which has higher security level.

The *-property is supposed to prevent declassification of information – no subject may write to an object, unless the object is properly secured. However, this is often considered rather a strong tool. If we take the military example into account, the *-property would disallow generals to write orders for privates, since such orders obviously have lower security level then the security levels of generals. Therefore, the concept of "trusted subjects" may be introduced into the system, allowing them to violate the *-property by declassifying objects and the *-property then being mandatory only for "untrusted subjects". Similarly, the "no write-down" rule may be omitted when the output (the written information) does not directly depend on the input (the classified information).

Moreover, the term s*trong *-property* is sometimes used to refer to such model, where a subject may write to an object only if $SL_o = SL_s$, i.e. no "write up" is allowed. This is motivated by integrity requirements, since it prevents less trusted subject from possibly corrupting more confidential data.

### 4.3.2 Biba

The previously discussed Bell-LaPadula model provided a way of preserving data confidentiality with the "no read-up, no write-down" rules. The Biba model [7] is

basically an inverse, substituting confidentiality for integrity, introducing "no read-down, no write-up" rules.

Let us define the *integrity level* the same way we defined the security level, i.e. tuple (compartments, classification) and let us also use the same definition of ordering. The model then proposes following properties:

- *The Simple Integrity Property* – for an integrity level $IL_s$ of a subject *reading* an object with a integrity level $IL_o$, it must hold that $IL_o \geq IL_s$. This is also known as the "no read-down" property.
- *The Integrity *-property* – for an integrity level $IL_s$ of a subject *writing* to an object with an integrity level $IL_o$, it must hold that $IL_o \leq IL_s$. This is also known as the "no write-up" property.

The goal of these properties is to prevent less trusted subjects corrupting data of more important objects and thus preserve their consistency.

## 4.4 Discretionary Access Control (DAC)

Unlike in MAC systems, where subjects were granted access to objects based on their security levels, in discretionary access control systems, the access is solely restricted based on the identity of the subject. Moreover, a subject can delegate its rights for certain object to another subject. The DAC model assumes existence of the *owner* of each object, who has unlimited rights to that object and is responsible for granting and revoking access of other subjects. [11]

A basic structure for managing DAC is an access control matrix, first proposed in 1973 by Lampson [12]. The matrix contains a row for each subject and a column for each object. The respective cells then contain lists of operations the given subject can perform on the given object.

Storing access rights directly in the matrix would be inefficient, since most of the cells would be blank. This has two possible solutions:

- Store only columns of the matrix, which for each object contain list of subjects and their privileges to that object. Such structure is then called an *Access Control List (ACL)*.
- Store the matrix rows, i.e. for each subject store the list of objects it can access and how. This is called a *Capability List*.

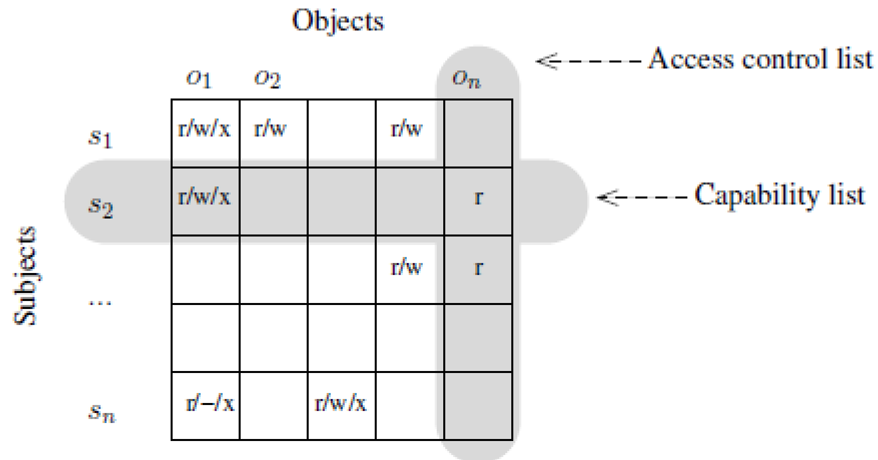


*Figure 5: Access control matrix [8]*

Each of these solutions has its disadvantages – for an ACL, the query "what objects can the given subject access" would be rather complicated to evaluate. Similarly when using Capability Lists, we cannot easily determine what subjects can access given object.

Despite this, DAC systems are practically used. The notorious example is the UNIX file system, which solves the problem of access control matrix size by reducing the number of subjects to three (owner, group and everyone else). The access rights for files (read, write, execute) are then stored using protection bits, effectively forming an ACL.

There has been some research regarding usage of DAC in a distributed environment. Since obviously neither access control matrix nor ACL are suitable for distribution, the basic idea in most of such systems is to authenticate remote users, map them to their local representatives and treat these as any other local subject. The strength of this approach then directly corresponds with the strength of the authentication system. The original cryptographic protocol proposed by Needham and Schroeder [13] had some weaknesses, but it laid the groundwork for additional research, which among others produced the well-known system Kerberos [14].

## 4.5 Role-Based Access Control (RBAC)

Utilizing DAC to provide data security stands on the premise that there is an *owner* among the subjects, who sets the security policy for the given object. It was observed in 1990s that this often does not hold – for example employees of a company access

and modify certain data, but the data itself belong to the company. To better model such situations, role-based access control models were researched and designed.

The basic elements of the RBAC models are:

- The set U of users – a user is typically a person within an organization, but it could also be a process, a device etc.
- The set P of permissions – a permission is an authorization to perform specific operation (read a file, invoke a method etc.). A permission is always positive (it allows rather than denies).
- The set R of roles – a role can be either viewed as a named collection of permissions or it can correspond with a job in the organization structure.
- $PA \subseteq P \times R$ – a many-to-many assignment of permissions to roles.
- $UA \subseteq U \times R$ – a many-to-many assignment of users to roles.

Users are granted with permissions only through roles, which provides much better control over the configuration of the system. Also, it is unlikely that any user would require all permissions for their work. Thus, the RBAC is often extended with the concept of *sessions*. A session is a mapping of one user to a subset of their roles. When established, a session intermediates all permissions from all roles in the session to the user. The previously mentioned *principle of least privilege* is therefore fulfilled.

Another RBAC extension is the *role hierarchy* model. This is obviously motivated by the user hierarchy in organizations, however, there are several interpretations of role hierarchy in RBAC. In one of them, role $R_1$ inherits role $R_2$ if all permissions assigned to $R_2$ are also assigned to $R_1$. In another, $R_1$ contains $R_2$ if all users assigned to $R_2$ are also assigned to $R_1$. And finally, $R_1$ could inherit $R_2$ if in all sessions where $R_1$ is active, $R_2$ is also active. In any case, role hierarchy aims to simplify the role management process by more accurately reflecting actual structure of an organization.

Organizations must often deal with the problem of *conflict of interests*. This stems from a situation where a single user is assigned conflicting roles, for example in a bank, a cashier should not be able to issue money to himself. In RBAC, it is possible to monitor roles being assigned to each individual user and set up such security policy that would prevent from one user having conflicting permissions. There are other security models focusing on the separation of duty, two of them – the Clark-Wilson

model and the Chinese Wall model – are discussed in the following sections. [15] [16] [8]

RBAC systems can be used in a distributed environment, as demonstrated on the dRBAC (Distributed RBAC) system [17]. This particular system uses distributed credential repositories called *wallets* to ensure that each component in the system can reason about providing access to confidential data. Since very similar attitude is used in DEECo, we consider distributed RBAC to be a good initial candidate for the devised security system.

## 4.6 Clark-Wilson

Similarly to the Biba model discussed in 4.3.2, the Clark-Wilson model [18] focuses on information integrity rather than confidentiality. It adopts principles and procedures from business and industry, particularly from banking systems. To ensure data integrity and consistency, the Clark-Wilson model formalizes the *separation of duty* principle (i.e. a user must be prevented from imposing conflicting roles) and the mechanism of *well-formed transactions* (i.e. any manipulation with data must leave the system in a consistent state). Formally, the model consists of the following basic elements:

- Constrained Data Items (CDIs) – the data items to which the integrity model must be applied.
- Unconstrained Data Items (UDIs) – the data items that are not under control of the model, e.g. user input or data from outside the system.
- Integrity Verification Procedures (IVPs) – when executed, an IVP verifies integrity of all CDIs in the system.
- Transformation Procedures (TPs) – correspond to well-formed transactions, which change a set of CDIs from one valid state to another valid state.

Only TPs can manipulate with CDIs. The model also assumes that before the first execution of a TP, the system was in valid state (because an IVP was executed). Then, by induction, the validity of the system is preserved even during repetitive execution of TPs.

While the system can ensure that only TPs manipulate CDIs, it cannot ensure that a TP will not corrupt the integrity of the system. For that, a TP must be certified to

implement certain transaction. The integrity policy can then be expressed in two types of formalization rules: the *certification* (which may have to be done manually and is application-specific) and *enforcement* (which is done automatically by the system and is application-independent).

The basic rules defined in the Clark-Wilson model are as follows:

- C1 (Certification): After running all IVPs, all CDIs must be in a valid state.
- C2: For each TP there must be a certification that the TP will preserve validity of all CDIs it processed.
- E1 (Enforcement): Only a TP can manipulate with a CDI.

These rules provide basic framework for maintaining consistency of CDIs. To provide a mechanism for also ensuring the *separation of duty* principle, we need to control which users execute the TPs:

- E2: A user can only perform those TPs on specific CDIs, for which they have an authorization.
- C3: The list of users and their authorizations from the E2 rule must be certified to meet the separation of duty requirement.

The E2 and C3 rules introduced the concept of user into the system. Since the user identity affects the TPs they can perform, it is necessary to perform an authentication. Thus:

- E3: The system must authenticate any user attempting to execute a TP.

To ensure data traceability and restorability, the Clark-Wilson model requires presence of an audit trail in the system. This can be modeled as another CDI and therefore only one more rule is needed:

- C4: All TPs must be certified to log the operation into and append-only CDI.

So far, the model only worked with CDIs. However, not all data in the system are CDIs – for example, new data entered by a user are inherently UDIs. It is therefore necessary to allow certain TPs to work with UDIs, producing a CDI on the output or raising an error:

- C5: Any TP taking an UDI as an input must be certified to either produce a CDI on the output or to perform no transformation at all.

Finally, this model would be useless if any user could modify their certifications. Therefore:

- E4: Only a super-user (administrator) may change authorizations to perform a TP.

Together, these nine rules define a model that enforces a consistent integrity policy. Later security models, such as the Chinese Wall model discussed in the following section, use the Clark-Wilson model as a groundwork for further enhancements. [18] [8]

## 4.7 Brewer-Nash (Chinese Wall)

Published in 1989, the Brewer-Nash model [19] (also known as the Chinese Wall model) represents the last step of transition from (military) data confidentiality to (commercial) data integrity and consistency. It aims to prevent the *conflict of interest* problem, which (as the authors argue) is as important to business as data secrecy to military sector. The Brewer-Nash model represents a business alternative to the traditional Bell-LaPadula model (discussed in 4.3.1).

The classical example on which the Brewer-Nash model is demonstrated consists of two business corporations and an accountant, working for both corporations. In such setup, it is necessary to prevent the accountant from taking advantage of their *insider knowledge* and using it against either of the corporations.

The corporate information is stored in a hierarchy. In the lowest level there are individual *objects* (items of information belonging to a single corporation). Objects of the same company are grouped into *company datasets* and in the highest level, datasets of corporations in competition are grouped into the *conflict of interest classes*. For convenience, let us mark the dataset of object $o$ as $y_o$ and its conflict of interest class as $x_o$.

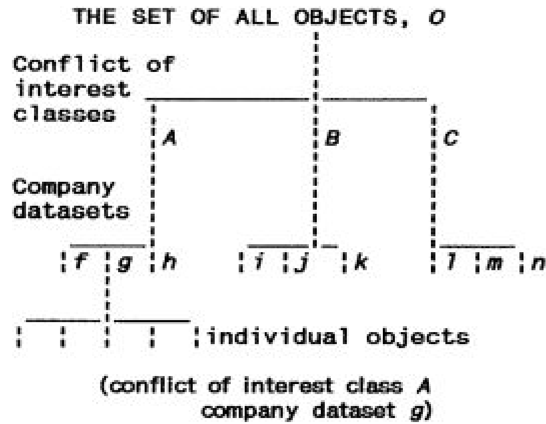


*Figure 6: The composition of objects in the Chinese Wall model [19]*

The basic idea of the Brewer-Nash model is the notion of *access history*. The conflict of interest prevention is achieved by allowing users to only access such data that are not in conflict with any other data they already possess. Formally, this is realized as a matrix $N$ with a row for each subject and a column for each object in the system. The value of $N_{s,o}$ is a Boolean value, which is true if and only if the subject $s$ has accessed object $o$.

Mirroring the Bell-LaPadula model, the Brewer-Nash model also defines the same two properties – the simple security property and the *-property (again read "star-property"). The simple security property states that an access to an object $o$ is granted only if one of the following conditions is satisfied:

- The requested object is in the same company dataset as previously accessed object, i.e. there is a previously accessed object $z$ such that $N_{s,z}$ = *true* and $y_z = y_o$.
- The requested object belongs to such conflict of interest class that the subject has never accessed before, i.e. for each object z that $N_{s,z}$ = *true*, it holds that $x_z \neq y_z$.

Obviously we presume that N was properly initialized, i.e. all values set to false. Nevertheless, the simple security property is not sufficient to prevent conflict of interests. Suppose there is an accountant Alice working for Microsoft and an accountant Bob working for IBM. Moreover, they also both work for Heineken (which obviously belongs to an entirely different conflict of interest class). However, there is nothing that would prevent Alice writing confidential information about Microsoft into the Heineken dataset for Bob to read.

The *-property prevents such indirect violations by introducing the concept of *sanitized data*. We consider data to be sanitized if it is not possible to discover or infer the identity of the corporation it came from. The process of disguising corporation's identity in data is called *sanitization*. The *-property than limits the access to object $o$ by specifying two necessary conditions that must be satisfied:

- Access to object $o$ must be permitted by the simple security property.
- There is no unsanitized object $z$ such that $N_{s,z} = true$ and $y_z \neq y_o$.

The second rule ensures that sanitized information can freely flow in the system, while unsanitized information is restricted to its dataset. [19] [8]

## 4.8 Distributed Security Models

The need for distributed security models came with the development of LAN networks and shared file systems. The security systems used locally did not scale well and often contained a single point of failure. Also, they lacked support for delegation as a way of decentralization and had little to offer in the fields of extensibility and expressibility.

We discussed one way of designing a distributed security system when describing distributed ACLs in section 4.4. This approach relied on subject authentication and its mapping onto a local identifier, which was then subjected to a standard local security policy. In this section, we describe security models based on capabilities and credentials, which are designed to scale better and provide more support for delegation of privileges.

### 4.8.1 Capability-Based Access Control

What among others prevents utilizing a local security model in a distributed environment is scalability. Even such a straightforward solution such as an ACL requires each subject's attempt to access an object to be validated by a central authority, thus creating possible performance bottleneck. Moreover, the ACL itself could be very long and therefore its checking would be inefficient. Combined with the lack of support for delegation, ACLs are not considered to be suitable for a distributed system.

Capability-based systems target to solve all these issues. A *capability* (sometimes referenced as a *key*) is a token granting permissions for a specific object to the

capability owner. Initially, the capability itself contains the object identifier and access rights. Since this would be vulnerable to tampering, the system associates every newly created object with a random secret. This random secret, the object identifier and the access rights are then used to create a hash, which is included in the capability. Formally:

$$hash = h(objectId, objectSecret, accessRights)$$
$$capability = (objectId, accessRights, hash)$$

Thanks to this, the system can at any time verify integrity of the capability by recomputing the hash. The capability does not contain any information about any subject, anyone who owns the capability is granted with the privileges the capability contains. This makes delegation of rights a very simple process – the delegator passes the capability he owns to the target subject, which can use it without any modifications. Also, revocation (i.e. removing existing access rights from subjects) can be easily achieved by resetting the object's secret, thus invalidating all capabilities associated with it.

However, capability-based approach in this setting does not prevent stealing of capabilities, their uncontrolled delegation nor duplication. It strongly relies on the *principle of least privilege*, but does not propose any way of ensuring it. In spite of that, the capability-based approach was even used with hardware support for example in the Cambridge CAP Computer [20] and it represents one of the most important developments in the distributed access control research. [21] [8]

### 4.8.2 Credential-Based Access Control

As discussed in the previous section, pure capabilities are not sufficient to provide reliable access control. A credential-based approach reuses the idea of a passable token serving as a tool for ensuring security policy. By a credential, we usually mean a statement made by an issuer about an identity or authority of a subject. This quite accurately reflects the real world, where for example citizens possessing a driving license are allowed to drive a vehicle. The issuer in this example a municipality, which are generally trusted to grant driving licenses only to people who can drive.

The notion of *trust* actually has great significance for the credential-based security systems, thanks to wide use of public-key cryptography. Usage of symmetric-key cryptography in distributed systems is complicated by the need of key management

and distribution. In asymmetric cryptography however, it is sufficient for a communication to only know the well-known public keys. Together with the concept of certificates (which bind the public key with some attributes of its holder and are signed by a *trusted* authority), they represent a widely used mechanism for access control and security enforcement.

Based on how credentials are used, distributed access control may be grouped into two categories: identity-oriented and key-oriented. [22]

#### *4.8.2.1 Identity-oriented*

This approach splits the access control into two stages – authentication and authorization. The purpose of authentication is to verify subject's identity by binding the specified public key to a name. Authorization then maps the name onto a set of privileges.

The well-known identity-oriented technologies are PGP (Pretty Good Privacy) and X.509 Public Key Infrastructure (PKI). Originally, the X.509 certificates only contained the subject's public key, DN (Distinguished Name) of the issuer, DN of the subject and validity information. The latest v3 version however allows to add *extensions*, thus enabling to add application-specific information. [23]

#### *4.8.2.2 Key-oriented*

The key-oriented approach merges authentication and authorization into a single step by omitting the usage of names. This solves the problem with scalability and flexibility of providing a unique name in a very large distributed system. Also, the name itself does not affect the process of controlling access, unlike in the real world, where we can decide based upon the subject's name whether we trust it.

Currently used example of key-oriented approach to credential-based security is the SPKI (Simple Public Key Infrastructure) [24]. [8]

# 5 Background: Trust Management

The term *trust management* was first defined in 1996 by Blaze et al. as *"a unified approach to specifying and interpreting security policies, credentials, relationships which allow direct authorization of security-critical actions"* [25]. Trust management systems were originally developed as an alternative mechanism for providing authorization, which would be better suited for large distributed systems, where individual security subjects are not known in advance and extensibility and expressibility are eminent. Examples of such systems are PolicyMaker and KeyNote, proposed by Blaze. We describe them in section 5.1.

While these systems focus on managing public key authorization, they ignore other aspects of trust. They do not provide any mechanism for trust relationships analysis, nor do they utilize experience of other subjects in own trust decision making. Another generation of systems was then proposed, defining trust management as *"the activity of collecting, codifying, analyzing and presenting evidence relating to competence, honesty, security or dependability with the purpose of making assessments and decisions regarding trust relationships for Internet applications"* [26]. An example of such system, SULTAN, is described in section 5.2.

## 5.1 PolicyMaker and KeyNote

While some of the security models described in chapter 4 attempt to adapt centralized approach to access control into distributed environment, such proposed systems do not cope well with at least one of the following aspects:

- Authentication – distributed access control must deal with the fact that subjects are not explicitly known at the time of creating security policy.
- Delegation – to prevent centralization, it must be possible for a subject to delegate its privileges to another subject (w.r.t. security policy).
- Expressibility and extensibility – any security policy must be possible to be modeled.

Blaze et al. [27] argue that rather than adopting centralized system, distributed security requires a brand new approach. In this section we briefly describe PolicyMaker and its successor, KeyNote, which were designed specifically for distributed environment needs.

The systems are built around a *trust management engine*, which is basically a query engine evaluating programmable request action against local security policy. It takes the following input: credentials presented by the requester, an action string and local security policies. The output can be either simple yes/no, or additional restrictions, which would make the request conform the local security requirements. Crucially, the action string format and local security policy specification format are general-purpose and application-independent, and therefore the PolicyMaker engine is capable of handling any incoming request.

The query has the following syntax:

$key_1, key_2, \ldots, key_n$ **REQUESTS** *ActionString*

The semantics of the *ActionString* is defined by the application, unlike the format, which is application-independent. The keys are the public keys identifying subjects issuing the request. Both policies and credentials are specified using assertions. Assertion is basically a statement delegating authorization from the signer to a subject, with the following syntax:

*Source* **ASSERTS** *AuthorityStruct* **WHERE** *Filter*

*Source* can be either the keyword **POLICY** in case of local policy assertions or a public key of a principal granting permissions in case of credentials. *AuthorityStruct* specifies a list of subjects to whom the assertion applies. *Filter* specifies a list of conditions the *ActionString* must meet to be considered valid.

Local policy assertions are only intended to be used inside the trust management engine and therefore are not signed. Credentials on the other hand are free to move in the system and therefore must be signed to protect their integrity. The set of policy assertions on a system forms a *trust root* (equivalent to a Certification Authority in X.509).

*Filters* are actually programs interpreted by the trust management engine. When a query is processed, the engine attempts to find a path from some trust root to the public keys requesting the action, where all the filters are satisfied. A filter works with an action string plus contextual information like date and time. This gives the application power to specify any security policy, without modifying or configuring the trust management engine.

The PolicyMaker allows three languages to be used in filters: AWK-WARD (safe version of AWK developed by the authors specifically for PolicyMaker), Java and

Safe-TCL[1]. This caused problems with interoperability, which was one of the reasons why KeyNote was created. KeyNote is a successor of PolicyMaker, but is more standardized and provides better integration into applications. It provides single, unified assertion language, which is designed to run smoothly with the trust management engine.

Both PolicyMaker and KeyNote are *assertion monotonic*, which means that negative assertions cannot be specified. There are certain trust management systems that allow negative assertions, e.g. REFEREE [28]. [8]

## 5.2 SULTAN

Trust management systems described in the previous section regarded trust as a way of executing access control. “To trust” in that context always meant “to provide access”. Grandison [29] however argues that authorization is just an outcome of a more abstract trust relationship and that trust does not imply access rights and vice versa.

Trust can then be regarded as “the quantified belief by a truster with respect to the competence, honesty, security and dependability of a trustee within a specified context”. This reflects the real world, where we (i.e. truster) always trust someone (i.e. trustee) about something (i.e. context) to some level (quantified). The proposed trust management model called SULTAN provides whole suite for specifying trust relationships, querying through Prolog, monitoring etc. Its primary purpose is to identify and analyze the effects of changing specifications on a business and to utilize these specifications to augment the security of Internet commerce. Unlike PolicyMaker or KeyNote described in the previous section, SULTAN supports also negative recommendations (i.e. its trust model is non-monotonic).

[1] http://www.tcl.tk/software/plugin/safetcl.html

# 6 Background: Cryptography

In this chapter we describe the very basic terms and principles of cryptography, which we later utilize when designing physical security for the DEECo data.

Cryptography can be defined as the science of *"processing data into unintelligible form, reversibly, without data loss"* [30]. Such techniques have been used long before computers had been invented, but for our purposes, let us focus on the digital cryptography, i.e. the problem of encryption and decryption of digital data. Cryptography is in that context used to solve the following issues that arise from exchanging messages between a sender and a recipient:

- Confidentiality – data are not accessible for unauthorized parties
- Integrity – it is impossible to tamper with data without the recipient noticing
- Authenticity – the recipient is able to reliably determine the sender of data
- Non-repudiation – the sender cannot deny sending the data

Before we continue with cryptography classification, let us define the basic terms:

- Plaintext – the readable data we want to secure
- Ciphertext – encrypted, unreadable data created by encrypting the plaintext
- Key – a token used to encrypt or decrypt the data (or to do both)

## 6.1 Symmetric Cryptography

In symmetric cryptography, the same key is used for both encryption of decryption of data.

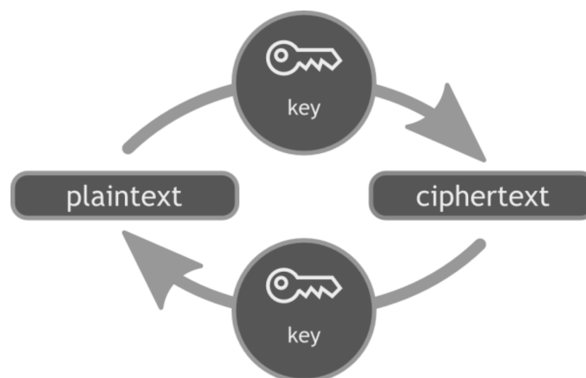


*Figure 7: Symmetric cryptography scheme [45]*

The obvious disadvantage of this concept is that before the two parties can communicate, they must first exchange the key securely. Although techniques exist that help establishing a secret key over a public channel (Diffie-Hellman key exchange [31]), there are still issues concerning authenticity, key exchange performance etc.

The main advantage of symmetric cryptography when compared to asymmetric cryptography is performance – symmetric algorithms are generally 1,000x – 10,000x faster [32]. Examples of such algorithms include DES (Data Encryption Standard), AES (Advanced Encryption Standard) or Blowfish.

## 6.2 Asymmetric Cryptography

Unlike symmetric cryptography, asymmetric cryptography (also known as public key cryptography) uses a pair of keys. One of them is called the public key and can be freely distributed. The other is called the private key and must be kept secret by its owner. To encrypt a portion of data, the sender uses the well-known public key of the receiver to create the ciphertext. However, only the private key corresponding to the encryption key can decrypt the data.

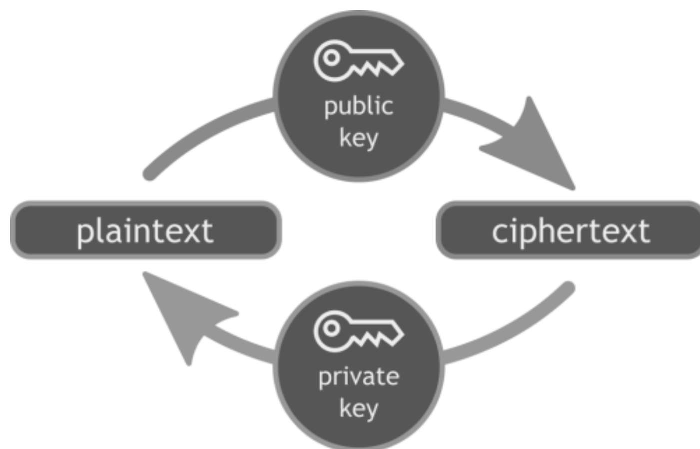


*Figure 8: Asymmetric cryptography scheme [44]*

This solves the main problem of symmetric cryptography – the exchange of the keys. The fact that someone knows the public key of a subject does not imply that they can decrypt its messages. However, a mechanism is still needed to securely obtain the public key for an entity. This is one of the tasks of the complex system called PKI (Public Key Infrastructure).

The main disadvantage of asymmetric cryptography is performance. Therefore, in practice, a hybrid encryption is used to merge benefits of both approaches to cryptography. We describe this in detail in section 8.3.2.

The most well-known example of an asymmetric encryption algorithm is RSA (Rivest, Shamir, Adleman).

## 6.3 Digital Signatures

Signatures are used to ensure data integrity and non-repudiation. They combine two mechanisms – hash (digest) functions and asymmetric cryptography.

Hash functions are used to irreversibly transform an arbitrarily long text into a fixed-length digest. This is not considered an encryption, since no key is used in the process and therefore anyone can create the same digest from the given data. The digest should depend on every bit of the input data and therefore it should not be possible to fabricate such data that would result in the given digest. Well-known hash functions include MD5, SHA etc.

Before sending the message, the sender computes a digest of the message using some hash function. This produces a fixed-length string, which the sender encrypts using their private key. The result is called a *signature* and is sent along with the rest of the message. The recipient of the message, who wants to make sure they obtained untampered data from the claimed sender, first uses the same hash function to compute a digest from the received message. Then the received signature is decrypted using the sender's public key. The result is compared with the computed digest and if they match, authenticity and integrity of the message is guaranteed.

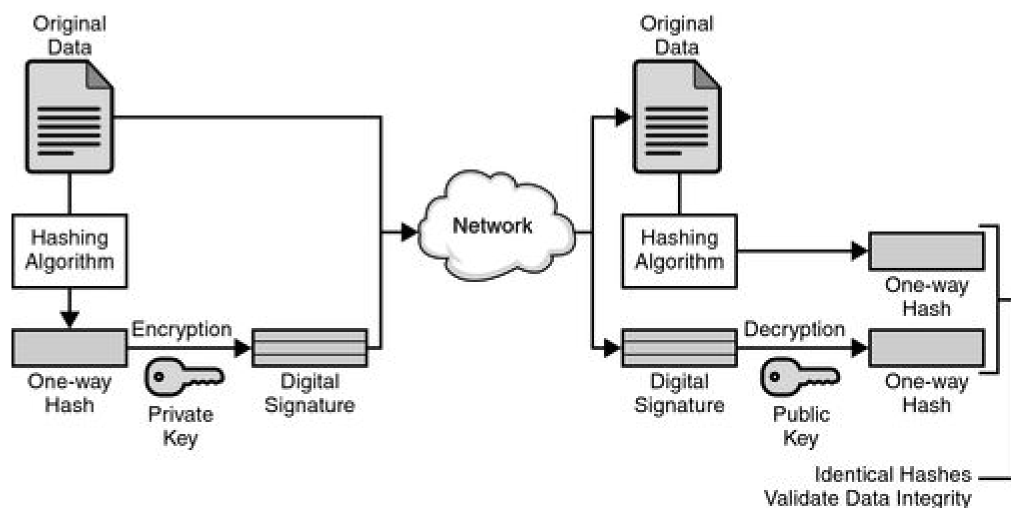


*Figure 9: Digital signature generation and verification scheme [43]*

# 7 Solution Strategy

In this chapter we utilize our knowledge of well-known security and trust models gained in chapters 4 and 5 to sketch a design of security and trust solution for DEECo, providing defense against threats described in chapter 3.

DEECo does not contain any central authority nor shared resources, apart from distributed knowledge. Also, a component has a very limited view of the system, specifically, it does not know other components in the system neither can it communicate with them.

We therefore take RBAC (section 4.5) as a groundwork for security, substituting DEECo components for subjects and setting up an association "component has-many roles". Next, we need to specify which roles (i.e. components) can access which part of knowledge. Inspired by ACL (section 4.4), we associate each knowledge path (i.e. security object) with roles that can access it. Since such list contains role identifiers and not directly component names, we avoid the problem with ACL length. To prevent data leakage, we take our inspiration from the Bell-LaPadula model (section 4.3.1) and we verify each knowledge exchange and component process to preserve security level (i.e. knowledge cannot lose security protection while being transferred). The roles associated with knowledge paths can be distributed along with the knowledge. The security is then enforced by every local runtime, when it compares the knowledge path roles and the local component roles and grants the component access only on successful match.

The trust part of the solution is mainly inspired by SULTAN (section 5.2), which suggests predicates "truster-trustee-object" to model trust relationships. Specifically, truster and trustee are components, object is a knowledge path. To create such relationships, we use the idea from the Clark-Wilson model (section 4.6) about IVPs (Integrity Verification Procedures). Each component can apart from standard processes also define a "rating process", through which it can check integrity of its data and provide rating of each knowledge path. This rating is then associated with the component the knowledge came from. By distributing such ratings among components, each component can at any moment check quality of its data and make respective adjustments.

# 8 Realization

In this section we describe the realization of the strategy from the previous chapter. We base our work on jDEECo, a Java implementation of DEECo developed in the Department of Distributed Systems and provide extensions for specification and enforcement of security policies.

## 8.1 Assumptions

When discussing runtime corruption (section 3.1.7) as a potential security threat, we mentioned the term TCB (Trusted Computing Base). TCB comprises all hardware and software, which we implicitly trust (usually because we have no other choice). In this very case, we consider the Java runtime, underlying operating system and hardware to be part of TCB. Also, we include the jDEECo runtime in TCB as well (of course without the definitions of components and ensembles – see 8.3.4). The last assumption however is added just in sake of simplicity – we could easily add a runtime signature hash to each message transferred between entities and instruct the recipient to verify that the sender is a proper jDEECo runtime.

Another assumption is related to the certificates and corresponding private keys used to encrypt transferred knowledge (as described in 8.3). We assume that all these certificates and keys are located in a keystore, which is copied to each instance of the jDEECo runtime and the runtime can access it. This is similar to assuming that all ensembles and components definitions are known to every runtime in the system.

Though this is not strictly required, we assume that the minimum version of Java runtime in which jDEECo is run is 1.8. Significant amount of new features has been added to the language[2], and some of them are utilized both in the implementation and in the examples in this text (for example repeatable annotations[3]). Even though a workaround exists for each new feature, we consider the new approach to be more intuitive.

## 8.2 Principles

In this section we define basic principles that we obeyed when designing the security and trust solution for DEECo.

---

[2] http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

[3] http://docs.oracle.com/javase/tutorial/java/annotations/repeating.html

1. The whole concept of security must be extensible. For example, there must be no such thing as a finite, hard-coded list of security roles.
2. The security policy must be defined declaratively and on the component level. The component itself knows best, how sensitive its data are.
3. The security always comes with a performance overhead. If no security is used by the components however, no overhead must be present. I.e. security must be an optional feature.
4. The basic communication paradigm of DEECo must not be changed (for example, components must not communicate directly).
5. The integrity constraints and the corresponding trust relationships utilization must be defined on the component level as well. Similarly to security, it is the component who knows the integrity constraints best.

## 8.3 Security Architecture

In this section we describe the mechanisms added to jDEECo to enable components protect their knowledge. Also, we describe how components themselves are verified before being deployed in the runtime.

### 8.3.1 Security Policy Specification

As we mentioned earlier, we use roles to restrict access to knowledge. Any component may be assigned (at design time) any number of roles. A role is defined by a Java interface annotated with *@RoleDefinition*, its assignment to a component is expressed by adding the *@HasRole* annotation to the component class definition. A State Police component from our running example (see 2.1) could be defined as:

```
@RoleDefinition
public interface PoliceRole {
}

@RoleDefinition
public interface IntegratedRescueSystemRole {
}

@Component
@HasRole(PoliceRole.class)
@HasRole(IntegratedRescueSystemRole.class)
public class StatePoliceVehicle {
}
```

*Figure 10: Simple roles assignment example*

Having added roles to components, we now need a mechanism for specifying the protection of their knowledge. Any knowledge field in the component definition may also be assigned any number of roles, thus instructing the runtime to allow only components owning the specified role to access the knowledge. This is achieved by adding the *@Allow* annotation.

```
@RoleDefinition
public interface AmbulanceRole {
}

@Component
public class OrdinaryVehicle {
	/**
	 * An unsecured field
	 */
	public String id;

	@Allow(value=PoliceRole.class,accessRights=AccessRights.READ)
	@Allow(value=AmbulanceRole.class,accessRights=AccessRights.READ)
	public String ownerName;

	@Allow(value=PoliceRole.class,accessRights=AccessRights.READ_WRITE)
	public boolean orderedToStop;
}
```

*Figure 11: Securing knowledge by specifying access roles*

In the example above, the *id* field is unsecured – any other component can read it. The *ownerName* field is accessible only for components possessing the police or ambulance role, the access is read-only in both cases (which means that such field cannot appear in the *@Out* or *@InOut* process/knowledge exchange parameter). The *orderedToStop* field can be both read and modified by the police components (and cannot be accessed by anyone else).

To improve expressibility, it is possible to use also the *@AllowEveryone* annotation, which is not parametrized with role, only with access rights. The purpose of this annotation is to specify basic security policy which applies to every component in the system. The following example demonstrates usage of the *@AllowEveryone* to specify that any component may read the data, but only Police may modify them.

```
@Component
public class OrdinaryVehicle {
	@AllowEveryone(AccessRights.READ)
	@Allow(value=PoliceRole.class,accessRights=AccessRights.WRITE)
	public boolean orderedToStop;
}
```

*Figure 12: Example with @AllowEveryone*

To make the role-based specification more dynamic, it is possible to add parameters to roles. For example, the Municipal Police component should have the police role, but only for the given city. It would be inconvenient to create a dedicated police role for each possible city and even more demanding to add them to protected knowledge fields. To address this, we propose three kinds of parameters:

- Parameter with absolute value
- Parameter referencing knowledge
- Parameter with wildcard

The following example shows examples for each kind of parameter:

```
@RoleDefinition
public interface PoliceRole_AbsoluteParam {
	@RoleParam
	public static final String cityName = "Prague";

	@RoleParam
	public static final Integer cityId = 123;
}

@RoleDefinition
public interface PoliceInCityRole {
	@RoleParam
	public static final String cityName = "[cityId]";
}

@RoleDefinition
public interface PoliceEverywhereRole
	extends PoliceInCityRole {
	@RoleParam
	public static final String cityName = null;
}
```

*Figure 13: Security role parameters example*

The parameter is defined as a public, static and final field in the role interface, annotated with *@RoleParam*. These modifiers are obligatory and are intended to prevent a component to modify its own security role parameters (as will be discussed in 8.3.4).

Parameter with absolute value can have any serializable type.

A path parameter is a string containing a knowledge path enclosed with brackets (to differentiate it from an absolute value parameter of type string). The knowledge path is evaluated each time the role is used and the obtained value serves as an actual parameter.

A wildcard parameter has a special significance when used together with role inheritance. Since security role is defined by an interface, inheritance can be used to achieve robustness and help maintainability. In the example above, the interface *PoliceEverywhereRole* inherits *PoliceInCityRole,* which implies that every component with the *PoliceEverywhereRole* role also has the *PoliceInCityRole*. It is possible to override parameters defined in roles parents. The wildcard parameter (which is recognized thanks to its *null* value) in this example models a situation where a component with *PoliceEverywhereRole* also has *PoliceInCityRole* for any *cityName* value.

### 8.3.2 Encryption and Signing

In the previous section we described how knowledge fields can be annotated in order to achieve security. In this section we focus on physical security, i.e. methods of protecting knowledge data by encryption and signing, to prevent unauthorized access and tampering.

First, let it be reminded that we considered the jDEECo runtime and underlying Java runtime, operating system and hardware to be part of TCB, i.e. we trust these components completely. Therefore, we consider objects stored in the runtime memory to be safe, protected by the operating system memory access control mechanisms etc. and thus without need of additional protection.

However, we must also protect the data when they leave the safe memory and are transferred to another entity in the system. jDEECo uses the *KnowledgeData* object as a serializable wrapper, containing (among others) the sender component ID, data version and the transferred knowledge (a map between a knowledge path and an object value). Several other items are present, but they are not concern to security (hop count and last sender ID used in rebroadcast, RSSI[4] etc.).

[4] Received Signal Strength Indication

Let us elaborate on possible security threats the *KnowledgeData* object may face during transmission. In the following sections, we describe protection mechanisms for each kind of security attack.

#### *8.3.2.1 Preventing Data Modification*

The attacker may attempt to capture transferred data and modify it. This does not necessarily imply that the attacker can read the data – enough damage can be done even with random modification. Also, the knowledge itself is not the only potential target. If the attacker for example set the data version to a lower value, the receiving component would consider such data obsolete and did not update its local knowledge. This way, the attacker could direct the flow of data in the system, which is highly undesirable. To ensure system integrity, we must prevent this even for unsecured knowledge (i.e. knowledge fields with no *@Allow* annotations).

We therefore propose for each *KnowledgeData* object to compute a hash, sign this hash with a dedicated private key and attach the signature to the *KnowledgeData* object. This way, the receiving entity can verify the integrity of the data by recomputing the hash and matching it against the decrypted signature. Also, it knows that the data came from a valid jDEECo instance (since no other would own the same key pair used for signing).

Not all items in the *KnowledgeData* object are signed – we exclude all items that are not subject to security and that may change during the transmission: hop count, last sender ID etc. Including them in the signature would create a performance issue since the signature would have to be recomputed each time the message is resent, excluding them on the other hand causes no security issue since they do not affect processing of received data.

#### *8.3.2.2 Concealing Data*

Thanks to a mechanism introduced in the previous section we can now be sure that messages cannot be modified along their way. However, we also need to be sure that knowledge data cannot be read. Moreover, we must deal with multiple *@Allow* annotations on a single knowledge field, meaning that any of given roles can access the field.

We could simply use a symmetric key and encrypt all secured knowledge data in the message. Since every key is known to every entity, the receiving side could decrypt the message, unlike the potential attacker who captured it. We however consider this

solution inappropriate, because symmetric keys are vulnerable to compromise and also the idea of every piece of data being encrypted the same way, regardless on the actual security level, seems wrong.

We therefore propose associating every *evaluated* security role with an asymmetric key pair and using this key pair for encryption. By "evaluated", we mean the security role whose parameter values were resolved to actual values – for absolute value parameters and wildcard parameters this is trivial, for path parameters we obtain the value from the component's knowledge. Thanks to this, the role "Municipal Police in Prague" (Prague being value of a parameter) will be associated with a different key then "Municipal Police in Pilsen". The sender entity then checks security roles associated with the knowledge being sent, obtains the respective public key (all keys are known to every entity, as we assume in 8.1), encrypts the knowledge and sends it. To make it possible for a receiving entity to get the private key, it is necessary to include the identifier of a role in the *KnowledgeData* object being sent. This identifier of course is included in the message signature to prevent its modification.

Several issues are introduced by this approach. First, there may be multiple *@Allow* annotations on a knowledge field, each of the roles of course associated with a different key pair. To address this, the sending entity splits the message being sent into several "submessages", each of these corresponding to a single security level. For example, consider the following component:

```
@Component
public class Component {
	public String field1;

	@Allow(value=RoleA.class)
	public String field2;

	@Allow(value=RoleA.class)
	@Allow(value=RoleB.class)
	public String field3;
}
```

*Figure 14: Example component to demonstrate submessages*

The first submessage would contain only the fields with no security protection, in our case only the *field1*, not encrypted. A submessage would then be created for each role used in the fields, i.e. a submessage containing *field2* and *field3* (both encrypted with the key corresponding to *RoleA*) and a submessage containing *field3*, encrypted with the *RoleB* key would be created.

The second and last issue concern the actual way of encrypting the knowledge data. The straightforward idea of simply encrypting the byte stream of the whole *KnowledgeData* object fails, because the object contains data that must not be encrypted – for example the security role identifier, last sender ID etc. Thankfully, Java offers the concept of *SealedObject*[5]. This is basically a safe created around any object we want to protect. Without a key to the safe (i.e. the key used to create the *SealedObject*), it is impossible to access it. Moreover, the *SealedObject* is serializable. We can therefore replace each protected knowledge object with a *SealedObject* containing it and pass the *SealedObject* to the lower layer of jDEECo that will take care of serialization and transmission. The receiving side then simply checks if the object it received is an instance of *SealedObject* and if so, it uses the security role identifier included in the message to obtain the key to it.

#### *8.3.2.3 Knowledge Path Risks*

As mentioned earlier, the *KnowledgeData* object that forms a message being transferred among jDEECo entities contains knowledge in the form of tuples (knowledge path, knowledge data). The mechanism of encryption described in the previous section encrypts only the data, not their knowledge paths. Could that be a potential security risk?

Since the attacker can read the knowledge paths, they may attempt to modify them. That would of course be a great security issue, however even though the knowledge paths are not encrypted, they are included in the signature hash and therefore the receiving entity would spot the inconsistency. This kind of attack is therefore not possible.

[5] http://docs.oracle.com/javase/7/docs/api/javax/crypto/SealedObject.html

Secondly, let us not forget on the concept of nested knowledge paths. We discussed them in section 2.5 – a knowledge path may contain nested paths, which are evaluated first and their value becomes part of the main knowledge path. For example, consider the following scenario:

```
@Component
public class MunicipalPolice {
	@Allow(value = PoliceRole.class)
	public Map<String, String> driversCredentials;

	@Allow(value = PoliceRole.class)
	public String pursuedDriverId;

	public MunicipalPolice() {
		this.pursuedDriverId = "123456/7890";
		this.driversCredentials = new HashMap<>();

		this.driversCredentials
			.put(pursuedDriverId, "John Smith");
	}
}
```

*Figure 15: Example of the Municipal Police component*

Now let us demonstrate the evaluation of the knowledge path *driversCredentials.[pursuedDriverId]*. First, the nested knowledge path *pursuedDriverId* is evaluated, which gives us *driversCredentials.123456/7890*. This path is then evaluated (i.e. a value corresponding to the *123456/7890* key is retrieved from the map) and we get the result: *John Smith*.

As we can see at the second step, the knowledge path at that moment actually contains knowledge data. If such single knowledge tuple was transferred in a message, it would look like this: (*driversCredentials.123456/7890, encrypted(John Smith)*). The actual value *John Smith* is properly encrypted for the *PoliceRole*, but the knowledge path is not – and the attacker may obtain an information that driver with ID *123456/7890* is being pursued. However, this situation can never appear in jDEECo, since only knowledge paths corresponding to whole knowledge fields are sent. In the example then, the knowledge paths *driversCredentials* and *pursuedDriverId* and corresponding encrypted values are sent. Even this kind of attack is therefore not possible.

#### *8.3.2.4 Performance*

As mentioned at the beginning of the section 8.3.2.1, in order to ensure data integrity, even messages containing no secured knowledge should be signed. This however conflicts with the Principle 3 discussed in section 8.2, because even though such component does not use any security feature of jDEECo, it is burdened with performance overhead caused by creating and verifying signatures. To address this issue, the signature generation for unsecured messages is an optional feature, turned on by setting the system property *deeco.security.sign_plaintext_messages* to *true*.

Also, asymmetric cryptography is known to be very slow. In the setup we described in section 8.3.2.2, we would use the public key of the security role to encrypt basically whole *KnowledgeData* object, which however can be potentially quite large and both its encryption and decryption therefore a performance issue. To address this problem, we use common solution for such situations – for each *KnowledgeData* object we encrypt, a random symmetric key is generated. This key is then used to encrypt the *KnowledgeData* (symmetric encryption is much faster). Then we use appropriate public key to encrypt just the symmetric key and we attach the result to the *KnowledgeData* object. Since the symmetric key is much smaller then whole *KnowledgeData*, the performance does not suffer that much. When a target entity receives such *KnowledgeData*, it uses the proper private key to decrypt the symmetric key attached in the message and this key then to decrypt the rest of the data. We therefore bypassed the problem with performance while preserving advantages of asymmetric encryption. This method is known as *hybrid encryption* [33].

### 8.3.3 Access Control

So far we described definition of the security policy (see 8.3.1) and mechanisms used to encrypt and sign data during their transmission (see 8.3.2). Now let us focus on the mechanism of access control, i.e. how the security policy is enforced and access to knowledge restricted.

The access control mechanism must be executed whenever a component obtains new data. In DEECo, this only happens in the knowledge exchange method of an ensemble. We must therefore introduce a new processing step – after a membership condition test, the runtime must verify that the local component owns such collection of roles that permits it to read the knowledge of the remote component. If and only if

both tests are passed (membership condition and security), the knowledge exchange can be performed.

First, let us identify the protected knowledge.

```
@Ensemble
@PeriodicScheduling(period = 5000)
public class PoliceRadar {

	@Membership
	public static boolean membership(
		@In("member.position") Coord memberPosition,
		@In("member.ownerName") String ownerName,
		@In("coord.position") Coord coordPosition) {
		// compute distance and return true if within range
	}

	@KnowledgeExchange
	public static void exchange(
		@In("member.ownerName") String ownerName,
		@InOut("coord.vehicleNearbyDriver")
			ParamHolder<String> vehicleNearbyDriver) {
		// use "member." to access member's knowledge
		// use "coord." to access coordinator's knowledge
		vehicleNearbyDriver.value = ownerName;
	}
}
```

*Figure 16: Example of an ensemble*

The membership condition may contain only *@In* parameters, we must therefore verify that the local component has read access to all knowledge paths in the parameters. The knowledge exchange method may contain any kind of parameters, we must verify all of those as well.

#### *8.3.3.1 Covert Channel Intermezzo*

Before we continue, let us discuss if it is really necessary to verify access rights to the membership method parameters. There cannot be any output parameters, so even if the component gained access to the knowledge it should not, the component has nowhere to write that information. Moreover, checking the parameters costs us time. However, this is a prime example of a covert channel (discussed in 3.1.6). Consider the following example of an "evil" ensemble:

```java
@Ensemble
@PeriodicScheduling(period = 1000)
public class EvilEnsemble {

	@Membership
	public static boolean membership(
		@In("member.pursuedCriminals") Set<String> pursuedCriminals)
	{
		return !pursuedCriminals.isEmpty();
	}
	@KnowledgeExchange
	public static void exchange(
		@Out("coord.policeInPursuit")
			ParamHolder<Boolean> policeInPursuit)
	{
		policeInPursuit.value = true;
	}
}
```

*Figure 17: Example of a covert channel ensemble*

If the membership condition parameters were not checked for access rights, this ensemble would leak information – any component could find out, it a nearby police is currently pursuing a criminal, even though they could not find out, which one.

#### *8.3.3.2 Forming an Ensemble*

To fully understand issues related with access control, we must first explain the jDEECo knowledge transmission paradigm in more detail.

Each jDEECo entity (i.e. each instance of the runtime) may contain multiple components. All of these components are considered *local*. Knowledge of these components is periodically sent to all other entities in the system via the *KnowledgeData* object. As we mentioned in section 8.3.2, the *KnowledgeData* object contains information about security role protecting its knowledge, so that the target component could determine the right key for decryption. Before the jDEECo security extension was introduced, each received *KnowledgeData* object updated a knowledge of a *replica* in the entity. A replica represented a copy of knowledge of a remote component, hosted on a local entity. An entity then contained a replica for each remote component in the system (from which it at least once received the *KnowledgeData* object). We will explain how this has changed by introducing a security model later in the section 8.3.3.3.

Forming an ensemble is then quite a simple process, in which the jDEECo runtime periodically iterates through all possible pair of local components and replicas and

attempts to form an ensemble. Inherently, there are two possible ways the ensemble may be formed:

1. Between two local components
2. Between the local component and a replica (of another remote component)

Considering forming an ensemble between two replicas is obviously pointless, since no knowledge of a local component would be updated.

For each tested pair of components, the jDEECo runtime first assigns the local component the role *coordinator* (the remote component thus becoming the *member*) and evaluates the membership condition. If that is passed, knowledge exchange is performed. Then, the jDEECo runtime switches the roles, the local component becomes a *member* while the remote one a *coordinator*. Again, membership condition is checked and possibly, knowledge exchange method invoked.

#### *8.3.3.3 Security Role Evaluation*

After a membership condition was passed, we need to check, whether the local component owns such set of roles that would enable it to read the knowledge mentioned in the arguments of the membership condition and knowledge exchange method. For that, two things are needed:

1. The security roles of the local component: since the local component has been annotated with *@HasRole*, we can also easily determine the roles.
2. The security roles that protect the knowledge fields of the remote component (i.e. were assigned using *@Allow*). However, to make this information available on the local component, it must be transferred from the definition of the remote component. Therefore we added it to the *KnowledgeData* object that is used to transfer data between entities (and encrypted the "security metadata" the same way as the knowledge data to prevent information leakage).

All the roles now must be evaluated, i.e. their parameters must be resolved to actual values. Especially the path parameters must be resolved using the respective component's knowledge. And at this point, we are facing two problems:

1. What if the path parameter used in the *@Allow* security role referred to *@Local* knowledge?

2. What if the knowledge path used in the ensemble methods arguments contained nested and protected knowledge paths?

Let us demonstrate the first problem on the following example.

```
@RoleDefinition
public interface PoliceRole {
	@RoleParam
	public static final String cityIdParameter = "[cityId]";
}

@Component
public class MunicipalPolice {
	@Allow(PoliceRole.class)
	public String pursuedDriverId;

	@Local
	public String cityId;

	public MunicipalPolice(String cityId) {
		this.cityId = cityId;
	}
}
```

*Figure 18: Ensemble using security role with @Local path parameter*

As we can see, the *pursuedDriverId* field is protected with the *PoliceRole*. However, in order to evaluate the role (i.e. resolve its parameters), the *cityId* field must be present in the knowledge as well. This causes no problem when evaluating the role locally, however, when this component becomes a replica (by transferring its knowledge to another entity), the *cityId* field will be missing, because it is decorated with *@Local* and therefore cannot be distributed.

To overcome this problem, we reutilize the encryption mechanism introduced in 8.3.2. We already included the security role information in the *KnowledgeData* object, so that the receiving component could determine the decryption key. Now we can use this already evaluated role information again for access control, rather than evaluate the role locally. To make this work, we however need to slightly modify the replica management. So far, the replicas were shared between the local components hosted on the same entity. This is no longer acceptable, since each component may have different access rights (i.e. different roles). Therefore, we propose a replica to be created for each local component individually, containing exactly those data that the local component can access. This way, whenever we check if a component has access to a knowledge of another component, we can always yield success when the second

component is a replica (because, essentially, the access control has already been performed when receiving the *KnowledgeData* object).

The second problem mentioned at the beginning of this section is related to the potential complexity of knowledge path used in ensemble method. As we described in section 2.5, a knowledge path may contain nested sub-paths, which are evaluated first and their value is used in their parent. Consider the following example:

```
@RoleDefinition
public interface PoliceRole { }
@RoleDefinition
public interface OfficerRole { }
@RoleDefinition
public interface CommanderRole { }
@RoleDefinition
public interface StateRole { }
@RoleDefinition
public interface CityRole { }

@Component
public class PoliceComponent {
	@Allow(PoliceRole.class)
	public Map<String, String> drivers;

	@Allow(StateRole.class)
	@Allow(CityRole.class)
	public Map<String, String> driverAges;

	@Allow(OfficerRole.class)
	@Allow(CommanderRole.class)
	public String wantedDriverId;
}

@Ensemble
public class ComplicatedEnsemble {
	@Membership
	public static boolean membership() { return true; }

	@KnowledgeExchange
	public static void exchange(
		@In("driverAges.[drivers.[wantedDriverId]]") String value,
		@Out("wantedDriverAge") ParamHolder<String> result
	) {
		result.value = value;
	}
}
```

*Figure 19: Example of complicated knowledge path in an ensemble*

Let us focus on the input parameter of the knowledge exchange method. Before actually executing the method, we need to check whether the local component has rights sufficient to access the knowledge specified by the path. Starting from the innermost path, the roles necessary for accessing the path are (we use abbreviations *PoR* for *PoliceRole*, *StR* for *StateRole* etc. to maintain readability):

1. *wantedDriverId*: `OfR OR CoR`
2. *drivers.[wantedDriverId]*: `PoR AND (OfR OR CoR)`
   using distributivity we get: `(PoR AND OfR) OR (PoR AND CoR)`
3. *driverAges.[drivers.[wantedDriverId]]*:
   `(StR OR CiR) AND ((PoR AND OfR) OR (PoR AND CoR))`
   using distributivity again:
   `(StR AND PoR AND OfR) OR (StR AND PoR AND CoR) OR`
   `(CiR AND PoR AND OfR) OR (CiR AND PoR AND CoR)`

As we can see, the set of roles necessary to access a knowledge path actually forms a DNF (Disjunctive Normal Form) formula, each literal representing a role the component must have. When comparing security level of knowledge (represented by its path) and a component, this formula has to be satisfied. Since satisfiability is an NP-complete problem (Cook-Levin Theorem [34]), we have no other choice but to test each of the disjuncts whether the accessing component has all the roles that match the roles specified by its formula literals.

To test if the two roles match, we simply need to do the following:

1. Test the role names for equality
2. Test the role arguments – for each argument of the *protecting* role (i.e. role assigned using @Allow), there must be an argument in the *accessing* role (i.e. role assigned using @HasRole) which:
   a. Has the same name
   b. Either has the same value or is *null* (as discussed in 8.3.1, null value acts as a wildcard)

#### *8.3.3.4 Indirect Access Control*

The mechanism of access control as we so far described it presumed that the access is provided based on the relation between the protecting component (which decorates its knowledge with *@Allow*) and the accessing component (which is decorated with *@HasRole*). However, this must not always be the case.

Let us return to the example with vehicles, Municipal Police (MP) and State Police (SP). The State Police role could be defined as “the Municipal Police role in every city”, as demonstrated on the following snippet:

```
@RoleDefinition
public interface MunicipalPoliceRole {
	@RoleParam
	public static final String cityIdParameter = "[cityId]";
}

@RoleDefinition
public interface StatePoliceRole extends MunicipalPoliceRole {
	@RoleParam
	public static final String cityIdParameter = null;
}
```

*Figure 20: Example of Municipal Police and State Police roles*

It could be desirable for a SP to share certain knowledge about ordinary vehicles only with MP that comes from the same city as the ordinary vehicle. That is, SP would obtain certain knowledge from ordinary vehicle, which is not accessible to MP. However, SP would annotate the knowledge in such way that it would be accessible for MP provided the city in which the MP belongs is the same as the city of the ordinary vehicle. This can be achieved using the evaluation context of the *@RoleParam*:

```
@RoleDefinition(aliasedBy = PoliceInAuthorsCity.class)
public interface MunicipalPolice {
	@RoleParam
	public static final String cityIdParameter = "[cityId]";
}

@RoleDefinition
public interface StatePolice extends MunicipalPolice {
	@RoleParam
	public static final String cityIdParameter = null;
}

@RoleDefinition
public interface PoliceInAuthorsCity extends StatePolice {
	@RoleParam(ContextKind.SHADOW)
	public static final String cityIdParameter = "[cityId]";
}

@Component
public class OrdinaryVehicleComponent  {
	public String cityId;

	@Allow(StatePolice.class)
	public String secret_for_city;
}

@Component
@HasRole(MunicipalPolice.class)
public class MunicipalPoliceComponent  {
	public String cityId;

	@Local
	public String secret_for_city;
}

@Component
@HasRole(StatePolice.class)
public class StatePoliceComponent  {
	@Allow(PoliceInAuthorsCity.class)
	public String secret_for_city;
}
```

*Figure 21: Example of an indirect access control*

First, let us focus on the *PoliceInAuthorsCity* role. The *SHADOW* context kind instructs the jDEECo runtime not to evaluate the parameter in the context of the component itself, but rather in the context of the component from the corresponding knowledge came from. When the role *PoliceInAuthorsCity* used to restrict access to *secret_for_city* is evaluated, jDEECo finds the replica from which this field was populated and resolves the *cityIdParameter* in its context. To determine the right replica, the concept of knowledge authorship is used (described in 8.4.3).

Secondly, the *aliasedBy* property of the *@RoleDefinition* annotation must be used. This simply adds the role of that name to the list of roles for given component.

The knowledge flow would then be:

1. The *StatePoliceComponent* obtains *secret_for_city* from the *OrdinaryVehicleComponent* (because the roles of *@Allow* and *@HasRole* match).
2. The *MunicipalPoliceComponent* attempts to access the same data, but fails (because the roles do not match).
3. The *MunicipalPoliceComponent* attempt to get the data from *StatePoliceComponent:*
    a. First, the role names for *secret_for_city* must match. Because the *MunicipalPolice* is aliased by *PoliceInAuthorsCity*, which is used in *@Allow*, the match succeeds.
    b. Second, the values of the *cityIdParameter* must be the same. In the *MunicipalPoliceComponent*, the parameter is easily evaluated by retrieving the value of the *cityId* knowledge path. In the *StatePoliceComponent*, the author of the *secret_for_city* knowledge is determined to be the correct *OrdinaryVehicleComponent* and the *cityId* knowledge path is evaluated in the context of its replica. Since these values will again match, the knowledge exchange will be performed.

Thanks to this mechanism, it is possible to create very specific security policies while maintaining the security level of the knowledge data. Also, we can perceive using the *SHADOW* context as a form of delegation.

### 8.3.4 Component Clearance Verification

Component's access privileges are determined by the security roles it owns. Since this is an essential part of the security model, we need to design a mechanism that would enforce the following requirements:

1. No component with security roles that may cause breaking the system security policy can be deployed.
2. Component cannot add or remove security roles of itself nor other components.
3. Component cannot modify any security role parameter values.

Let us start with the first requirement. It is rather vaguely formulated – since each component is an autonomous unit, the phrase "system security policy" cannot be

defined explicitly, but rather as "union of security policies of all components". In our running example, the ordinary vehicle component marked certain knowledge to be accessible only for components with the police role. The protecting component then relies on the system no to contain a fabricated "bad component" with the police role. We need a way of maintaining such setup, where each component has only those roles that it is safe to be entrusted with.

For that, we introduce the concept of *certification authority* (CA). Similarly to PKI (Public Key Infrastructure), this authority is inherently trusted by all components and its job is to verify that new components meet the security policy requirements. The process of deploying a new set of components is then:

1. Develop (code) the new components and package them in a JAR (Java Archive) file.
2. Send this JAR to a well-known certification authority.
3. The authority verifies that the components in the JAR conform the security policy. If the verification succeeds, the CA signs the JAR.
4. When jDEECo loads the JAR containing the components, the signature is checked and only if it is proved to be a valid signature of the CA, the components are loaded and deployed.

Thanks to this, we can rely on the CA to do the (possibly manual) verification of security role usage. Since this checking is only performed in design-time, we have also bypassed the problem of creating a performance bottleneck.

To simplify the process, jDEECo does not verify the JAR signature in case no component in the JAR owns any security role. In that case, we can deploy the components without consulting the CA, without risking security breach. Moreover, to ease development, it is possible to turn off signature checking altogether by setting the system property *deeco.security.verify_secured_component_jars* to *false*.

The second requirement from the beginning of this section stated that no component can add or remove security roles of itself or other components. The security roles are assigned to a component when it is deployed, jDEECo provides no mechanism of modifying the collection of security roles at runtime. Also, no component is allowed to communicate with any other component directly. Thanks to these two facts, it is guaranteed that the collection of security roles of a component remains intact.

Finally, the third requirement says that component cannot modify the parameter values of the security roles it owns. For instance, the Municipal Police role from our running example has a parameter identifying the city, in which this role is valid. Obviously, it is highly undesirable to let any component modify such value. Before we propose a mechanism that would prevent such situations, let us describe the problem with respect to security role parameter definition introduced in 8.3.1.

There are three kinds of security role parameters:

- *Absolute*, containing any Java object or primitive value
- *Path*, containing a string knowledge path
- *Blank*, containing Java *null*, which acts as a wildcard

```
@RoleDefinition
public interface TestRole {
	@RoleParam
	public static final Integer cityId = 123;

	@RoleParam
	public static final String cityReference = "[cityId]";

	@RoleParam
	public static final String cityWildcard = null;
}
```

*Figure 22: An example role containing all kinds of parameters*

As we can see, it would be very easy for a component to modify the parameter value, if the modifier *final* was not used. The *final* keyword guarantees that:

- Absolute parameters containing primitive types or Strings are read only (because these types are immutable)
- Path parameters are read only (since they are always Strings).
- Wildcard parameters are read only (because once *null* is assigned, it cannot be replaced with any other value).

However, the *final* keyword will not prevent a situation where the value of an absolute parameter is an object. For example, consider the following:

```java
@RoleDefinition
public interface TestRole2 {
	@RoleParam
	public static final TestObject testObject = new TestObject();
}

public interface TestObject {
	public int x = 3;
}
```

*Figure 23: Example of a security role containing object parameter value*

Despite the *final* keyword, anyone can modify the value of the field *x* and therefore modify the parameter of the security role. To prevent this, all security parameter values are cloned before being loaded into the jDEECo runtime and these clones are not accessible from component code. Thus, even if someone modifies the value of *x,* the cloned value and also the security role remains unchanged.

Finally, we need to address the issue with path security role parameters. Even though the knowledge path itself is set as a compile-time String constant, the knowledge data (that are later used as the actual parameter of the role, when the knowledge path is evaluated) can potentially be modified, for example in the component processes.

To solve this, all knowledge paths used in the security role parameters are marked as *locked*. Whenever a method is called that is allowed to modify knowledge (i.e. component process or knowledge exchange), its output arguments are checked for being locked and if they are, the method is not called. It is not sufficient to perform his check just once when the component is deployed, since the knowledge paths in the output arguments may contain evaluable nested paths (which may result in different absolute knowledge path on each evaluation).

### 8.3.5 Data Leakage Prevention

So far we described mechanisms used to protect knowledge and control access of components to the knowledge. In this section, we introduce the problem of data leakage and present a suitable solution.

Let us consider the following component:

```
@RoleDefinition
public interface Role {}

@Component
@HasRole(Role.class)
public class LeakingComponent {
	@Allow(Role.class)
	public String protectedKnowledge;

	public String unprotectedKnowledge;

	@Process
	public static void process(
		@In("protectedKnowledge") String protectedKnowledge,
		@Out("unprotectedKnowledge") ParamHolder<String>
unprotectedKnowledge
	) {
		unprotectedKnowledge.value = protectedKnowledge;
	}
}
```

*Figure 24: Example of data leaking component*

The *LeakingComponent* from the example above clearly has access to both *protectedKnowledge* (because it has the necessary role) and *unprotectedKnowledge*. However, each time the *process* is invoked, the confidential knowledge is copied into a freely distributable knowledge field, the knowledge thus losing its protection. Such behavior is obviously undesirable, we need to make sure that whenever a component is entrusted with certain knowledge, it will not "betray" us and make the knowledge available for someone without proper security clearance.

To address this issue, we propose a "data leakage" verification to be performed before each knowledge exchange. This is the only moment when a component obtains a new knowledge and therefore the only moment that is critical to data leakage. The verification procedure needs to check the target component (i.e. the component whose knowledge is being modified) if there is no situation, where a piece of knowledge would be copied from more secured field to a less secured field. To model the dependencies between knowledge fields, the jDEECo *@In*, *@InOut* and *@Out* parameter kinds are used. Specifically, the algorithm of the verification runs as follows:

1. Iterate through all processes of the target component
    a. For each input parameter (i.e. decorated with *@In* or *@InOut*), create transitive closure of dependent parameters

2. For each such parameter and its dependencies, evaluate the security roles protecting its knowledge path (see 8.3.3.3 how security protection of a knowledge path actually forms a logical formula in DNF)
3. Verify that no output parameter has lower security level than the original input parameter.

If this check does not yield success, the knowledge exchange is not performed and the knowledge security therefore is not jeopardized.

Specific case is knowledge annotated with *@Local* which depends on secured knowledge. In such situation, the verification process can yield success immediately, because local knowledge is never distributed to other components and therefore the data cannot leak.

Verification of data leakage as we just described it is rather a strong tool – it never allows any output parameter to be less secure then any input parameter, despite the semantics of the data. Let us discuss one example, where such strict behavior is undesirable (we already mentioned it briefly in 3.1.6):

```
@RoleDefinition
public interface PoliceRole {}

public enum SirensState { ON, OFF }

@Component
@HasRole(PoliceRole.class)
public class PoliceWithSirens {
	@Allow(PoliceRole.class)
	public String pursuedCriminal;

	public SirensState sirensState;

	@Process
	public static void determineSirensState(
		@In("pursuedCriminal") String pursuedCriminal,
		@Out("sirensState") ParamHolder<SirensState> sirensState
	) {
		if (pursuedCriminal == null) {
			sirensState.value = SirensState.OFF;
		} else {
			sirensState.value = SirensState.ON;
		}
	}
}
```

*Figure 25: Example of a component process with inevitable data leakage*

In this example, the police vehicle component contains a process which decides whether the vehicle sirens should be on or off. This decision is made according to a

value of secured *pursuedCriminal* field. However, the data leakage verification process would detect that the unsecured *sirensState* fields depends on secured *pursuedCriminal* field and therefore this component would not receive any data in the knowledge exchange method, even though apparently no unnecessary knowledge is made public.

To solve this issue, it is possible to decorate any process method with @*IgnoreKnowledgeCompromise*. When this annotation is applied, the process method is not checked for data leakage, enabling components to mark harmless dependencies. It is a job of CA (described in 8.3.4) to make sure this annotation is not abused to arbitrarily copy data between fields with incompatible security levels.

Lastly, let us consider performance footprint of the data leakage prevention mechanism we just described. Before each knowledge exchange, it is necessary to get dependency graph for each knowledge field in the target component, evaluate corresponding roles and check if no field loses its security level. Using simple benchmarking jDEECo simulation and VisualVM[6] as a profiler, we estimated the effect of data leakage check is 20% slowdown. Therefore we introduce a caching layer, which stores results of data leakage checks and thus keeping the performance footprint minimal. By introducing such layer, the slowdown caused by repetitive execution of data leakage checks in removed entirely.

## 8.4 Trust Architecture

In this section, we describe the trust model proposed for utilization in DEECo along with realization in jDEECo. Its goal is to enable components reason about quality of their knowledge, possibly preventing violation of integrity constraints.

### 8.4.1 Concept

As mentioned in chapter 7, the proposed trust model takes inspiration from the SULTAN system described in 5.2. To reflect also the real world, where trust is always quantified, we propose to store trust relations as quaternions:

- The truster, i.e. the ID of the component which is the source of the trust relationship

[6] http://visualvm.java.net/

- The trustee, i.e. the ID of the component with which is the truster in relationship
- The knowledge path, i.e. what the truster trusts the trustee with
- The rating, i.e. a value from selected enumeration of possible states of the knowledge (e.g. *OK*, *UNUSUAL* etc.)

These quaternions are distributed across the components the same way as knowledge and are therefore available at any component at any time. Because the quaternion contains a concrete knowledge path (which must be absolute, i.e. not to contain nested knowledge paths), it is possible to rate knowledge at any level of detail.

The quaternions are created by the components themselves through dedicated process (see 8.4.3), idea being that it is the component who knows the integrity constraints the best. It is inspired by IVP (Integrity Verification Procedure) as defined in the Clark-Wilson model (see 4.6).

### 8.4.2 Obtaining the Rating

To make ratings accessible, a new method parameter kind is introduced, called @*Rating* (adding to already existing @*In*, @*InOut* and @*Out*). This parameter kind can be used in any method – component process, ensemble membership condition or knowledge exchange and takes a knowledge path as an argument as well. Such parameter must have the type *ReadonlyRatingsHolder*, which provides method to get the number of components that rated the knowledge path with the specified rating level. The @*Rating* parameter usage is demonstrated on the following example:

```java
@Component
public class ComponentWithRating {

	public double outsideTemperature;

	@Process
	public static void process(
		@In("outsideTemperature") double outsideTemperature,
		@Rating("outsideTemperature") ReadonlyRatingsHolder 
outsideTemperatureRating
	) {
		if (outsideTemperatureRating.getRatings(PathRating.OK)
		    > 2) {
			// treat outsideTemperature as reliable
		} else {
			// treat outsideTemperature as unreliable
		}
	}
}
```

*Figure 26: Example of component using knowledge rating*

In this example, the number of components that rated the knowledge path *outsideTemperature* with value *OK* is obtained. This is the only functionality the *ReadonlyRatingsHolder* provides, specifically, it does not enable to modify any ratings.

Since the ratings data are aggregated and do not contain any knowledge or component information, these data are not subjected to access control. Security roles and data leakage prevention described in section 8.3 are therefore not applied, any component can obtain rating for any knowledge path. To prevent information leakage through covert channels (for example by test-rating a knowledge path to check if it contains data), the rating mechanism does not even check if the knowledge path being rated exists.

### 8.4.3 Creating the Rating

It is the component who knows best what integrity constraints should its knowledge meet, therefore it should again be the component who provides rating of the knowledge data. To satisfy this requirement and also to satisfy the need to prevent components from setting the ratings arbitrarily, we propose the ratings to be set in a dedicated process of the component.

Any component may define a single *ratings process*, which is a method similar to common component process. This process however cannot contain any output parameters (*@InOut* nor *@Out*), to prevent knowledge modification. On the other

hand, it may contain *@Rating* parameters with type *RatingsHolder* (as opposed to *ReadonlyRatingsHolder* described in previous section).

This object provides functionality for setting the rating of the knowledge path, as demonstrated on the following example:

```
@Component
public class ComponentWithRating {

	public double outsideTemperature;

	@RatingsProcess
	public static void ratingProcess(
		@In("outsideTemperature") double outsideTemperature,
		@Rating("outsideTemperature") RatingsHolder
outsideTemperatureRating
	) {
		if (outsideTemperature > -50 && outsideTemperature < 50) {
			outsideTemperatureRating
				.setMyRating(PathRating.OK);
		}
		else
		if (outsideTemperature > -100 && outsideTemperature < 100) {
			outsideTemperatureRating
				.setMyRating(PathRating.UNUSUAL);
		} else {
			outsideTemperatureRating
				.setMyRating(PathRating.OUT_OF_RANGE);
		}
	}
}
```

*Figure 27: Example of a component rating process*

The method providing the knowledge rating is decorated with *@RatingsProcess*, there can be at most one such method defined per component. In the example above, the ratings process takes the knowledge of path *outsideTemperature* and based on the value, sets the rating for this path.

The ratings process of a component is (if present) called after each successful knowledge exchange, since it is the only moment where the component knowledge is enriched with outside data that may potentially violate integrity constraints.

As we mentioned earlier, the ratings form a quaternion. So far the following is obvious:

- The truster component is the component whose rating process is invoked
- The knowledge path is the path from the *@Rating* parameter
- The rating is the value of enumeration set in the *setMyRating* method of the *RatingsHolder* object

However, we also need to know the trustee, i.e. the component whose knowledge is being rated. For that, we need to introduce the concept of *knowledge authors*.

When the component is first deployed in the jDEECo runtime, it is also set as the author of its whole knowledge. The only moment when the authorship of knowledge can change is during knowledge exchange. At this time, the authors of the knowledge are updated the same way as the knowledge itself, i.e. the author of the given knowledge path from source component is also set as the author of the knowledge path at the target component.

If a piece of knowledge is modified in the component process (i.e. the *@InOut* or *@Out* parameter was used), the component itself is set as the author of corresponding knowledge paths (since it may have changed the knowledge data). However, if the component did not modify the knowledge it received, the author remains the same as set in the knowledge exchange, even when this knowledge is redistributed. It is therefore necessary to include the knowledge authors in the *KnowledgeData* object that is used to send data between components. The information about knowledge authors is of course encrypted the same way as the knowledge data itself.

Thanks to this concept, the jDEECo runtime can always for the given component and knowledge path determine ID of the component, where the knowledge came from. In the *ratings process*, this is used to determine the trustee, i.e. the component whose knowledge is being rated. It is also used in all other *@Rating* parameters to initialize the *ReadonlyRatingsHolder* (i.e. to determine how is the given knowledge path of the given component rated).

### 8.4.4 Ratings Distribution

As we described earlier, component knowledge is periodically distributed to other components, where it forms a *replica*. The *KnowledgeData* object is used as a wrapper for transferring the knowledge itself, security attributes (see 8.3.3.3), knowledge authors (see 8.4.3) and other metadata.

Similarly, rating information is distributed using the *RatingsData* object, which contains list of quaternions created during invocation of the *rating processes* of the local components. Since the ratings are potentially confidential, the *RatingsData* content is encrypted with well-known key to prevent compromise during transmission. The ratings data are also distributed periodically, along with the knowledge data.

# 9 Evaluation

In this chapter we describe the implementation of the running example which utilizes the proposed security mechanisms. Then we profile the simulation to monitor performance footprint.

The simulation uses two kinds of vehicles: the police vehicle and the ordinary vehicle. The ordinary vehicle behavior is the same as in [2], i.e. each vehicle starts at certain location, is assigned certain destination and uses means described in [2] to reach it.

The police vehicle contains the same logic, i.e. it too travels between two locations. However, each police vehicle also contains a list of names of wanted criminals. While travelling, police vehicle uses the ensemble called *PoliceRadar* to monitor ordinary vehicles within range and if any of them is driven by a wanted criminal, the police vehicle starts a pursuit by setting itself the same destination as the pursued vehicle. Let us see the corresponding code (certain parts of it have been removed for brevity):

```
@Component
public class Vehicle {
	/** Id of the vehicle component. */
	public String id;

	/** Destination place */
	public String dstPlace;

	/** Position of the current link. */
	public Coord position;

	/** Contains a list of link ids that lead to the destination. */
	@Local
	public List<Id> route;

	public VehicleKind vehicleKind = VehicleKind.ORDINARY;

	@Allow(value = PoliceRole.class,accessRights = AccessRights.READ)
	public String ownerName;

	...
}
```

*Figure 28: Ordinary vehicle component used in simulation*

```java
@RoleDefinition
public interface PoliceRole {    }

@Component
@HasRole(PoliceRole.class)
public class PoliceVehicle {
	/** same fields as in ordinary component – id, route,
	destination, speed … */

	@Local
	public String[] wantedOwnerIds;

	@Local
	public String currentlyPursuedOwnerId, currentlyPursuedVehicleId;

	public VehicleKind vehicleKind = VehicleKind.POLICE;

	@Local
	public Map<String, String> vehiclesOwnersNearby,vehicleIdsNearby;

	...
}
```

*Figure 30: Police vehicle used in simulation*

```java
@Ensemble
@PeriodicScheduling(period = 1000)
public class PoliceRadar {
	@Membership
	public static boolean membership(
		@In("member.position") Coord memberPosition,
		@In("member.vehicleKind") VehicleKind memberKind,
		@In("coord.position") Coord coordPosition,
		@In("coord.vehicleKind") VehicleKind coordKind) {
		return
Navigator.getEuclidDistance(memberPosition,coordPosition) <=
Settings.POLICE_RADAR_RANGE &&
		memberKind == VehicleKind.ORDINARY &&
		coordKind == VehicleKind.POLICE;
	}

	@KnowledgeExchange
	public static void exchange(
		@In("member.id") String id,
		@In("member.ownerName") String ownerId,
		@In("member.dstPlace") String dstPlace,
		@InOut("coord.vehiclesOwnersNearby")
ParamHolder<Map<String, String>> vehiclesOwnersNearby,
		@InOut("coord.vehicleIdsNearby") ParamHolder<Map<String,
String>> vehicleIdsNearby) {
		vehiclesOwnersNearby.value.put(ownerId, dstPlace);
		vehicleIdsNearby.value.put(ownerId, id);
	}
}
```

*Figure 29: PoliceRadar ensemble used in simulation*

As we can see, the *PoliceRadar* ensemble adds data about an ordinary vehicle (member) to a police vehicle (coordinator). The police vehicle than uses these data in a dedicated process, which consults the map of vehicles nearby with the given list of wanted criminals and if a match is found, the police vehicle sets its destination to the destination of the pursued vehicle (if only life was so simple…).

The testing simulation includes 40 ordinary vehicles and 10 police vehicles, 10 minutes of service is simulated. To implement the lower communication and simulation levels, MATSim[7] and OMNet[8] are used. Since jDEECo is not entirely deterministic, each of the experiments below is run 10 times, the hosting computer (4 core Intel i7, 6GB memory available for the JVM, Windows 8.1) was restarted after each run. Slight inaccuracy is introduced by the Java JIT (Just-In-Time) compiler – the measured times do not include just the simulation, but also the time needed to compile the Java bytecode to a native language. But since this time is more or less constant for each experiment and run, we consider this inaccuracy irrelevant.

The following experiments were conducted:

A. Completely removed security, i.e. the *@Allow* annotation protecting the *ownerName* field is removed and messages are not signed.
B. Protected the *ownerName* field with the *@Allow* annotation as listed in Figure 28, but turned off the plaintext message signing.
C. Removed the *@Allow* annotation, but signing of messages enabled.
D. Utilized both *@Allow* annotation and plaintext messages signing.

The results of the experiments are summarized in the following tables and graph:

*Table 1: Time in seconds required to run simulations*

| Run / Exp. | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **A** | 601.19 | 596.76 | 607.10 | 624.90 | 602.98 | 608.77 | 592.00 | 602.16 | 618.40 | 599.65 |
| **B** | 913.55 | 917.93 | 921.13 | 919.38 | 922.17 | 921.95 | 916.69 | 895.65 | 918.93 | 905.58 |
| **C** | 642.36 | 632.67 | 642.72 | 642.56 | 677.31 | 642.62 | 694.24 | 645.88 | 637.29 | 650.32 |
| **D** | 976.53 | 971.25 | 997.94 | 980.37 | 997.77 | 974.51 | 986.37 | 948.24 | 943.66 | 992.90 |

[7] http://www.matsim.org/

[8] http://omnetpp.org/

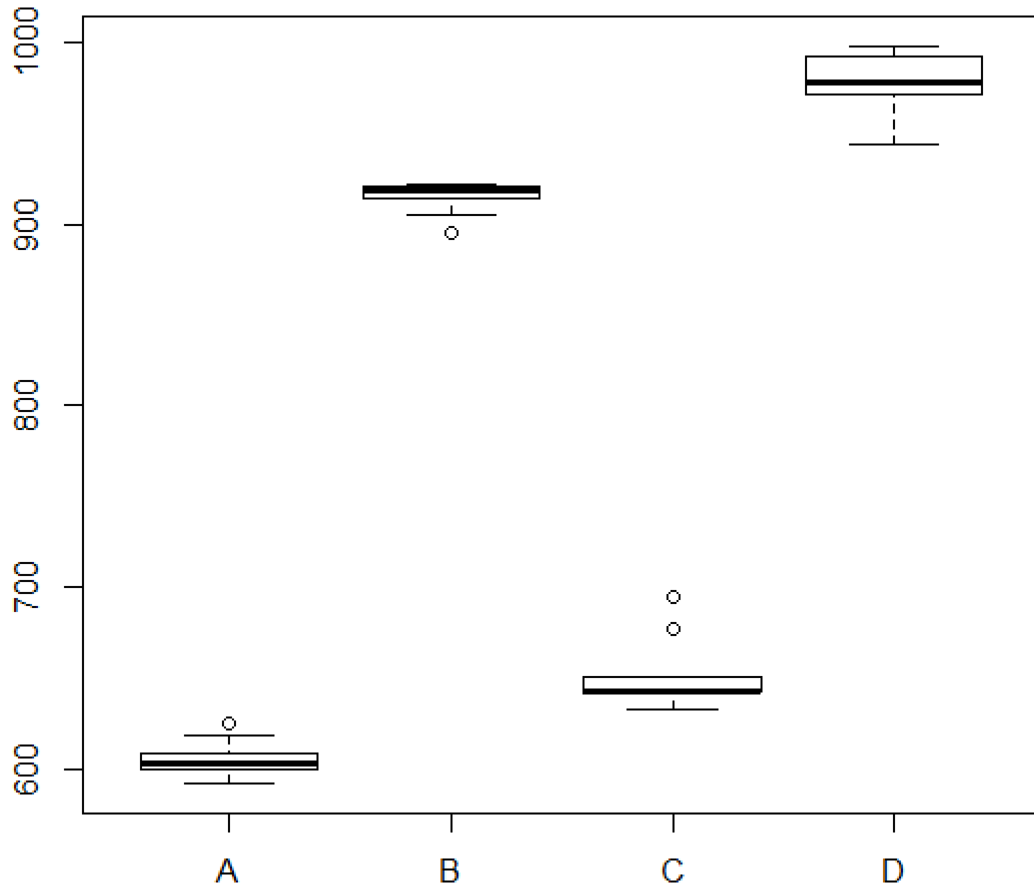


*Figure 31: Box plot displaying time elapsed during simulations of setups A - D*

*Table 2: Number of messages sent during experiments*

| Run / Exp. | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **A** | 7184818 | 7186331 | 7185581 | 7185767 | 7185103 | 7185261 | 7184132 | 7186402 | 7185241 | 7184772 |
| **B** | 8616110 | 8616306 | 8613529 | 8615357 | 8616350 | 8616265 | 8615556 | 8615016 | 8615233 | 8617015 |
| **C** | 7185302 | 7185277 | 7185277 | 7185921 | 7184373 | 7186089 | 7185501 | 7185258 | 7184663 | 7186086 |
| **D** | 8616473 | 8615946 | 8616718 | 8615019 | 8614814 | 8613162 | 8614897 | 8615992 | 8614107 | 8617032 |

The increase of number of messages between experiments with security (B, D) and without security (A, C) is caused by sending protected data in separate messages, as explained in 8.3.2.1. To find out the cause of elapsed time increase, we use VisualVM to find out that most of the extra time is spent encrypting messages and generating signatures for such messages.

Interestingly, VisualVM shows that almost all of the delay is caused by using public key cryptography to encrypt symmetric keys and generate signatures. The performance footprint of symmetric encryption is barely recognizable. On average, the increase in number of messages is 20%, the extra time is 51%.

# 10 Discussion

In this chapter we review the security model proposed in chapter 8 from various points of view and discuss its several drawbacks. Specifically, we discuss whether the result system complies with principles we set ourselves in section 8.2.

Most importantly, the solution does not contain any runtime centralized logic, which would be highly unsuitable for any CPS. Even though in 8.3.4 we introduced the concept of certification authority, this is not an issue since the authority is only consulted at deploy-time, not at runtime.

The model we propose is fully extensible – a new security role can be created without having to redeploy other components. Also, thanks to the concept of security role parameters, the expressibility of such solution is very high.

Classic subject authentication as we know it from the Bell-LaPadula model for example (see 4.3.1) is not present in our solution. Instead, security roles are used to classify components. Thanks to this, a component does not need to know the identity of all components with which it may share an ensemble and therefore exchange data.

The security roles are also used for authorization, as described in 8.3.3.3. Thanks to role inheritance and the concept of parameters, the model can well reflect the real world needs.

Delegation of rights is achieved using the indirect access control, as described in 8.3.3.4. This enables even more dynamic exchange of data.

However, it is not possible to modify access rights at runtime, i.e. perform revocation of privileges. For example, once deployed, a component cannot change the set of roles for which certain knowledge is accessible. This is caused by the fact that the whole component definition is loaded by the jDEECo runtime when the component is deployed and the component has no way of modifying such setup afterwards. The only way of revoking access to knowledge is to undeploy the component, perform the changes in security specifications and redeploy the component. This issue could however be easily solved – for each part of the knowledge, we could also store the ensemble through which it has been obtained. By periodically re-evaluating the ensemble's membership condition, we could revoke the rights the component once already had.

Knowledge in DEECo is protected by specifying roles which can access it. As we mentioned in chapter 7, this approach is inspired by classical ACLs, where instead of

specifying concrete subjects, more general security roles are used. However, this causes the DEECo solution to share also the drawbacks of ACLs. Namely, the query: “What knowledge can the given security role access?” would be hard to evaluate, since it would lead to iterating across all deployed components. Fortunately, such query is never needed in standard DEECo usage.

# 11 Related Work

In this chapter we compare our solution of DEECo security with similar systems that provide access control and trust management for decentralized systems.

## 11.1 dRBAC

Freudenthal et al. [17] built their dRBAC (Distributed Role-based Access Control for Dynamic Coalition Environments) system around a similar concept of security roles as DEECo. Their solution focuses among other things on third-party delegation of rights. Unlike in DEECo, assignment of security roles is not signed by a globally trusted certification authority, but rather by any other subject in the system. This concept allows subjects not only to be granted certain roles, but also to be granted a right to delegate certain roles to others. While this is more dynamic and provides better support for delegation, to verify subject's access rights, it is necessary to backtrace the delegation chain and perform verification on each element, which can be a performance issue. The dRBAC systems targets this problem by introducing the proof monitors (an entity can be notified about changes in the delegation chain) and caching of delegations. The security rights of certain subject are stored locally in a repository called the *wallet*. Each wallet maintains a consistent view of the credentials in the system using delegation subscriptions. The wallet is therefore quite similar to knowledge in DEECo, the difference being that a wallet contains only security information, whereas knowledge represents all data accessible to a component.

## 11.2 RT Framework

The RT framework [35] addresses the problem of decentralized access control by introducing a combination of RBAC, trust management and logic programming. Similarly to DEECo, the purpose of RT roles is to avoid the issue of individual subjects having to know all the others in order to enforce a security policy. From trust management, RT takes principles of managing distributed authority through the use of credentials. The languages from the RT family are based on DATALOG [36], a restricted form of logic programming. These languages are used for identifying subjects, specifying policy statements and queries.

The most basic language of the family is $RT_0$, where roles are specified simply by their names. In $RT_1$ however, a concept of role parameters is introduced, which is very

similar to DEECo. A role parameter can either be a constant (which corresponds to an absolute security role parameter in DEECo) or a variable (corresponding to the path parameter in DEECo). $RT_1$ also supports a variable to be specified a value set, which allows more flexible policy then the DEECo blank security role parameter.

The $RT^T$ language extends the $RT_0$ with tools to enforce a *separation of duty* (SoD) principle among the security subjects. Because DATALOG is used to assign roles to subjects, it is possible to enforce the SoD principle on the language level, unlike in DEECo, where it is a responsibility of a certification authority. Lastly, the $RT^D$ language is able to handle delegation of role activation. All the languages can be combined, the $RT_1^{DT}$ thus being an ultimate language containing features of all the others. The RT framework also contains an infrastructure necessary for utilizing the framework in real-world environment. Similarly to dRBAC, this includes a goal-oriented credential chain discovery and support for policy statement creation, storage and distribution.

## 11.3 IoT Security

When designing a security solution for the Internet of Things (IoT), several additional issues need to be addressed. Most importantly, devices in IoT may have very limited resources – memory, power, storage etc. In such environment, even a standard HTTP protocol is too complex and new dedicated protocols such as CoAP [37] are therefore being designed to access and control the devices.

Several access control models are built over such protocols. Hernández-Ramos et al. [38] propose their solution based on capabilities (see 4.8.1), which is designed to be as simple and as power-saving as possible.

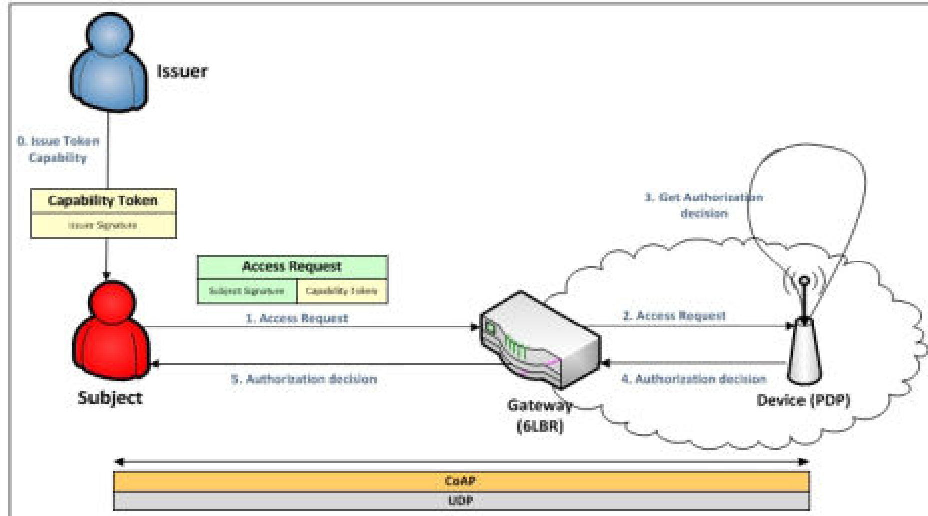


*Figure 32: Distributed Capability-based approach to IoT [38]*

When a subject wants to perform certain operation on an object, it must first obtain corresponding Capability Token from an Issuer. The Token contains information about the object, the operation to be performed and additional conditions to be checked on the target device. To prevent tampering, the Token is signed. The Subject then issues the request to the object and attaches the corresponding Token – the device hosting the Object simply checks if the request complies with the Token and eventually performs the operation.

While this approach can be simpler than our solution of security in DEECo, it is primarily designed for situations where the target device knows all potential subjects that may access it. Such assumption is legitimate in certain use cases (e.g. smart home can be controlled only by family members), but in others can be rather problematic.

# 12 Conclusion and Future Work

In this thesis we proposed an extension for the DEECo component model that would satisfy the need for security and trust management. We started by describing DEECo and its current Java implementation, jDEECo. Based on the DEECo architecture and principles, we analyzed potential security threats and the role of trust in the component model.

Next chapters were dedicated to research on existing and well-known security models. We concluded that some of them (e.g. the Reference Monitor discussed in 4.2.1) were designed specifically for usage in operating and other highly centralized systems and therefore cannot be taken as a source of inspiration for our work. As the security models developed, some of them became applicable even for distributed systems – particularly the RBAC (Role-Based Access Control) family of models does not rely on any centralized authority and thus it became the groundwork for our proposed security systems.

An example of such security model, the Clark-Wilson model (see 4.6), inspired us to define knowledge integrity through rating processes in components. Thanks to this concept, the component can rate knowledge it owns and the jDEECo runtime then aggregates (based on the authorship of the knowledge) the ratings among the components.

We then described our solution in detail and evaluated its usability on a Java implementation of the running example. We noticed the performance overhead introduced mainly by encrypting and decrypting messages and considered this to be a potential extension point.

Eventually we compared our work with related research, discussing the complexity of adding particular features to our solution.

Since jDEECo is still being developed in the Department of Distributed and Dependable Systems, the security layer will need to be developed as well. The currently proposed functionality includes option to build jDEECo without any security level to simplify development and also to rate data based on multiple versions, not just a single snapshot. Also, the concept of specifying value sets for security role parameters introduced in $RT_1$ would be useful in the DEECo security model.

# 13 Bibliography


[1] Venkatasubramanian, Krishna Kumar. *Security Solutions For Cyber-Physical Systems.* s.l. : Arizona State University, 2009.

[2] Bures, Tomas; Gerostathopoulos, Ilias; Hnetynka, Petr; Keznikl, Jaroslav; Kit, Michal; Plasil, Frantisek. *DEECo – an Ensemble-Based Component System.* s.l. : ACM, 2013.

[3] Gordon, Peter. *Data Leakage – Threats and Mitigation.* s.l. : SANS, 2007.

[4] Belinda, Fairthorne. *OMG White Paper on Security.* s.l. : OMG Security Working Group, 1994.

[5] Bowden, Joel S. *Security Policy: What it is and Why.* s.l. : SANS, 2003.

[6] Bell, David Elliot; LaPadula, Leonard. *Secure computer systems: Mathematical foundations.* 1973.

[7] Biba, K. J. *Integrity Considerations for Secure Computer Systems.* 1977. MTR-3153.

[8] Yao, Walt. *Trust management for widely distributed systems.* s.l. : University of Cambridge, 2008. ISSN 1476-2986.

[9] Hewlett Packard. HP OpenVMS Guide to System Security. *HP OpenVMS Systems Documentation.* [Online] June 2010. http://h71000.www7.hp.com/doc/84final/ba554_90015/ch02s01.html.

[10] Kleidermacher, David and Kleidermacher, Mike. *Embedded Systems Security.* s.l. : Newnes, 2012. ISBN: 978-0-12-386886-2.

[11] Schneider, Fred B. Discretionary Access Control. [Online] 2012. https://www.cs.cornell.edu/fbs/publications/chptr.DAC.pdf.

[12] Lampson, Butler W. *Protection.* s.l. : SIGOPS, 1973.

[13] Needham, R. M. and Schroeder, M. D. *Using encryption for authentication in large networks of computers.* 1978.

[14] Steiner, J. G., Neuman, C. and Schiller, J. I. *Kerberos: An authentication service for open network systems.* s.l. : USENIX Association, 1988.

[15] Ferraiolo, David F. and Kuhn, D. Richard. *Role-Based Access Controls.* s.l. : 15th National Computer Security Conference (1992), 1992.

[16] Sandhu, S. Ravi, et al. *Role-Based Access Control Models.* s.l. : IEEE Computer, 1995.

[17] Freudenthal, Eric, et al. *dRBAC: Distributed Role-based Access Control for Dynamic Coalition Environments.* s.l. : IEEE, 2002.

[18] Clark, David D. and Wilson, David R. *A Comparison of Commercial and Military Computer Security Policies.* s.l. : IEEE, 1987.

[19] Brewer, David F. C. and Nash, Michael J. *The Chinese Wall Security Policy.* s.l. : IEEE, 1989.

[20] Levy, Henry M. *The Cambridge CAP Computer.* 1988.

[21] Levy, Henry M. *Capability-Based Computer Systems.* s.l. : Digital Press, 1984.

[22] Camenisch, Jan, et al. *Credential-Based Access Control Extensions to XACML.* s.l. : W3, 2009.

[23] Curry, Ian. *Version 3 X.509 Certificates.* s.l. : Entrust Technologies, 1996.

[24] Ellison, Carl M. SPKI/SDSI Certificates. [Online] January 24, 2004. [Cited: 2 2, 2015.] http://world.std.com/~cme/html/spki.html.

[25] Blazen, Matt, Feigenbaum, Joan and Lacy, J. *Decentralized Trust Management.* s.l. : IEEE, 1996. ISSN 1081-6011.

[26] Grandison, Tyrone. *Trust Specification and Analysis for Internet Applications.* s.l. : Imperial College of Science, Technology and Medicine: London, 2001.
[27] Blaze, Matt, et al. *The Role of Trust Management in Distributed Systems Security.* s.l. : Springer, 1999.
[28] Chu, Yang-Hua, et al. *REFEREE: trust management for Web applications.* s.l. : Elsevier Science, 1997.
[29] Grandison, Tyrone and Sloman, Morris. *Trust Management Tools for Internet Applications.* s.l. : University of London, 2003.
[30] Schulzrinne, Henning. *Introduction to Cryptography (Lecture Notes).* s.l. : Columbia University .
[31] Diffie, Whitfield and Hellman, Martin E. *New Directions in Cryptography.* s.l. : IEEE, 1976.
[32] Knotek, Miroslav. PKI. *MS Fest Prague.* 2015.
[33] Mateescu, Georgiana and Vladescu, Marius. *A Hybrid Approach of System Security for Small and Medium Enterprises: combining different Cryptography techniques.* s.l. : IEEE, 2013.
[34] Cook, Stephen A. *The complexity of theorem-proving procedures.* s.l. : ACM, 1971.
[35] Ninghui, Li and Mitchell, John C. *RT: A Role-based Trust-management Framework.* s.l. : Standford University, 2003.
[36] McCarthy, Jay. Datalog: Deductive Database Programming. [Online] [Cited: 27 2, 2015.] http://docs.racket-lang.org/datalog/.
[37] Shelby, Z., et al. *Constrained Application Protocol (CoAP).* s.l. : IETF, 2013.
[38] Hernández-Ramos, José L., et al. *Distributed Capability-based Access Control for the Internet of Things.* s.l. : Journal of Internet Services and Information Security (JISIS), volume: 3, number: 3/4, pp. 1-16.
[39] Bures, Tomas; Gerostathopoulos, Ilias; Hnetynka, Petr; Keznikl, Jaroslav; Kit, Michal; Plasil, Frantisek. *Gossiping Components for Cyber-Physical Systems.* 2014.
[40] Wilbur, Steve. Distributed Systems Security (Lecture Notes). s.l. : University College London, 2008.
[41] Benes, Antonin. Information Security (Lecture Notes). s.l. : MFF UK, 2011.
[42] Sirer, Emim Gun. *Security Models (Lecture Notes).* s.l. : Cornell University, 2007.
[43] Oracle Corporation. How Directory Server Provides Encryption. *Oracle Directory Server Enterprise Edition Reference.* [Online] Oracle, 2011. [Cited: 2 20, 2015.] http://docs.oracle.com/cd/E20295_01/html/821-1222/gbgic.html.
[44] *Wikimedia Commons.* [Online] [Cited: 3 1, 2015.] http://commons.wikimedia.org/wiki/File:Orange_blue_public_key_cryptography_en.svg.
[45] *Wikimedia Commons.* [Online] [Cited: 3 1, 2015.] http://commons.wikimedia.org/wiki/File:Orange_blue_symmetric_cryptography_en.svg .
[46] Georges, Andy, Buytaert, Dries and Eeckhout, Lieven. *Statistically Rigorous Java Performance Evaluation.* s.l. : ACM, 2007.

# 14 List of Figures

Figure 1: Example of the State Police component ........................................................ 4
Figure 2: Example of ensemble - police radar ............................................................ 5
Figure 3: Example component for demonstrating knowledge path variants ............... 6
Figure 4: The Reference Monitor model schema........................................................ 13
Figure 5: Access control matrix [8] ........................................................................... 17
Figure 6: The composition of objects in the Chinese Wall model [19] ..................... 22
Figure 7: Symmetric cryptography scheme [45]......................................................... 29
Figure 8: Asymmetric cryptography scheme [44] ...................................................... 30
Figure 9: Digital signature generation and verification scheme [43]......................... 31
Figure 10: Simple roles assignment example............................................................. 34
Figure 11: Securing knowledge by specifying access roles........................................ 35
Figure 12: Example with @AllowEveryone ............................................................... 36
Figure 13: Security role parameters example............................................................. 36
Figure 14: Example component to demonstrate submessages ................................... 39
Figure 15: Example of the Municipal Police component........................................... 41
Figure 16: Example of an ensemble............................................................................ 43
Figure 17: Example of a covert channel ensemble ..................................................... 44
Figure 18: Ensemble using security role with @Local path parameter ...................... 46
Figure 19: Example of complicated knowledge path in an ensemble......................... 47
Figure 20: Example of Municipal Police and State Police roles................................. 49
Figure 21: Example of an indirect access control ...................................................... 50
Figure 22: An example role containing all kinds of parameters ................................ 53
Figure 23: Example of a security role containing object parameter value................. 54
Figure 24: Example of data leaking component ......................................................... 55
Figure 25: Example of a component process with inevitable data leakage ............... 56
Figure 26: Example of component using knowledge rating ...................................... 59
Figure 27: Example of a component rating process.................................................... 60
Figure 28: Ordinary vehicle component used in simulation ...................................... 62
Figure 29: PoliceRadar ensemble used in simulation ................................................ 63
Figure 30: Police vehicle used in simulation ............................................................. 63
Figure 31: Box plot displaying time elapsed during simulations of setups A - D ..... 65
Figure 32: Distributed Capability-based approach to IoT [38].................................. 71

# 15 List of Abbreviations

| | |
|---|---|
| ACL | Access Control List |
| CA | Certification Authority |
| CPS | Cyber-Physical System |
| DAC | Discretionary Access Control |
| DEECo | Distributed Emergent Ensembles of Components |
| EBCS | Ensemble-Based Component Systems |
| IoT | Internet of Things |
| JIT | Just-In-Time |
| JVM | Java Virtual Machine |
| MAC | Mandatory Access Control |
| OMG | Object Management Group |
| PKI | Public Key Infrastructure |
| RBAC | Role-Based Access Control |
| TCB | Trusted Computing Base |
| TCSEC | Trusted Computer System Evaluation Criteria |

# 16 Attachments

The directory on the enclosed CD has following structure:

- */doc*
    - */thesis* – contains the PDF version of this document
    - */generated* – documentation generated by Doxygen[9]
- */src* – contains source files of jDEECo. Most of the code related to this thesis is located in the two packages:
    - *cz.cuni.mff.d3s.deeco.security*
    - c*z.cuni.mff.d3s.deeco.integrity*

  These packages are located in the */src/jdeeco-core/src/* and */src/jdeeco-core/test/* folders. The following packages were also modified:
    - *cz.cuni.mff.d3s.deeco.annotations*
    - *cz.cuni.mff.d3s.deeco.annotations.processor*
    - *cz.cuni.mff.d3s.deeco.knowledge*
    - *cz.cuni.mff.d3s.deeco.network*
    - *cz.cuni.mff.d3s.deeco.task*

  All edited source files are decorated with the *@Author Ondřej Štumpf* declaration.
- */README.txt*

[9] http://www.stack.nl/~dimitri/doxygen/

# 17 Appendix – Build Instructions

To build the jDEECo projects, 64bit Java 1.8[10] and Maven[11] are used. Use the `mvn clean install` command in the *src/jdeeco-parent* folder for building and running the tests.

Alternatively, all subfolders of the *src* directory are Eclipse[12] projects. It is therefore possible to build and run tests using the IDE, which must have the Maven plugin[13] installed. First, create an Eclipse workspace (make sure *not* to choose the *src* directory as the workspace location) and import all three projects from the *src* directory (using File → Import → General → Existing Projects into Workspace). After the import, run the Maven update command using Alt + F5 (with the options *Clean projects* and *Force Update of Snaphots/Releases* selected). Now all standard build and test facilities of Eclipse are available. Let us remind that Java 1.8 is required – project *Build Path* may need to be edited if it was installed later on.

To run the simulations described in chapter 9, slightly extended version of the simulation runner used at the Department of Distributed and Dependable Systems was used. To get it, use Git to fetch the *police-simulation* branch from https://github.com/ostumpf/cbse-2015-tutorial . Import the obtained project into the Eclipse workspace, add the *jdeeco-core* and *jdeeco-simulation* projects as references and run the Maven update command again. Now create a *Java Application* run configuration in Eclipse with the following properties:

- Main class: “cz.cuni.mff.d3s.roadtrain.demo.SimulationRunner”
- Program arguments: “police 40 10 1” (simulation kind identifier, ordinary vehicles count, police vehicles count, run)
- VM arguments: “-Dlog4j.configuration=log4j-custom.xml -Xmx6096m”, possibly adding “-Ddeeco.security.sign_plaintext_messages=true”

After running the configuration, the elapsed time and number of messages sent is printed to console.

[10] http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

[11] https://maven.apache.org/

[12] http://www.eclipse.org/luna/

[13] http://eclipse.org/m2e/