Deliverable D5.3
Integrated Approach for Enforcing End-User Privacy Policies across Third-party Cloud Infrastructures
Release 1.0

The research leading to these results has received funding from the European Community’s H2020 research and innovation programme under grant agreement n° 731678.
<table>
<thead>
<tr>
<th><strong>Project Number</strong></th>
<th>731678</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Project Title</strong></td>
<td>RestAssured - Secure Data Processing in the Cloud</td>
</tr>
<tr>
<td><strong>Deliverable Type</strong></td>
<td>External Report</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Title of Deliverable</strong></th>
<th>Integrated Approach for Enforcing End-User Privacy Policies across Third-party Cloud Infrastructures</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Nature of Deliverable</strong></td>
<td>External Report</td>
</tr>
<tr>
<td><strong>Dissemination Level</strong></td>
<td>Public</td>
</tr>
<tr>
<td><strong>Contractual Delivery Date</strong></td>
<td>31 October 2019</td>
</tr>
<tr>
<td><strong>Actual Delivery Date</strong></td>
<td>31 October 2019</td>
</tr>
<tr>
<td><strong>Contributing WPs</strong></td>
<td>WP5</td>
</tr>
<tr>
<td><strong>Editor(s)</strong></td>
<td>Zoltan Mann</td>
</tr>
</tbody>
</table>
## Contributors

<table>
<thead>
<tr>
<th>Name</th>
<th>Organization</th>
<th>Sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoltan Mann</td>
<td>UDE</td>
<td>1, 6</td>
</tr>
<tr>
<td>Florian Kunz</td>
<td>UDE</td>
<td>2, 4</td>
</tr>
<tr>
<td>Jan Laufer</td>
<td>UDE</td>
<td>2, 4</td>
</tr>
<tr>
<td>Julian Bellendorf</td>
<td>UDE</td>
<td>2, 4</td>
</tr>
<tr>
<td>Andreas Metzger</td>
<td>UDE</td>
<td>5</td>
</tr>
</tbody>
</table>
## Document History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>V0.1</td>
<td>September 2019</td>
<td>Initial version</td>
</tr>
<tr>
<td>V0.1</td>
<td>October 2019</td>
<td>Final version based on cross-work-package review</td>
</tr>
</tbody>
</table>
Contents

List of Figures ........................................................................ 7
List of Tables ......................................................................... 9
1 Introduction ...................................................................... 10
2 Overview .......................................................................... 11
  2.1 Embedding in the RestAssured architecture .................. 11
  2.2 From design time to run time .................................... 12
  2.3 The run-time activities .............................................. 14
3 The use of models@run.time ............................................. 16
  3.1 Meta-model ................................................................ 16
    3.1.1 Design considerations for the meta-model .......... 16
    3.1.2 Initial meta-model .......................................... 17
    3.1.3 Structure of the meta-model ........................... 17
    3.1.4 Contents of the sub-models ............................. 18
    3.1.5 Updated meta-model ..................................... 22
  3.2 Run-time model .......................................................... 23
4 Detection and mitigation of data protection violation risks ....... 25
  4.1 Risk patterns ............................................................... 25
  4.2 Implementation of the risk pattern approach .................. 26
  4.3 Empirical results ........................................................ 29
  4.4 Mitigation of data protection violation risks ................... 30
5 Optimization-based approaches .......................................... 32
  5.1 Handling colocation risks in cross-layer cloud optimization 32
    5.1.1 Aspects of the component–VM–PM mapping .......... 33
    5.1.2 Problem formulation ....................................... 35
    5.1.3 Mapping algorithms ....................................... 37
    5.1.4 Evaluation ..................................................... 43
    5.1.5 Summary of findings ...................................... 46
  5.2 Optimized secure deployment of software components in hybrid clouds 47
    5.2.1 Problem .......................................................... 48
    5.2.2 Algorithm .......................................................... 52
6 **Outlook: Ensuring data protection in fog computing** ........................................... 62

6.1 **Data protection challenges in fog computing** ............................................. 62

6.2 **The case for adaptive data protection** .................................................... 63

   6.2.1 **Enabling adaptations** ..................................................................... 63
   6.2.2 **Adaptation in end devices** ............................................................... 64
   6.2.3 **Adaptation in fog resources** .......................................................... 64

6.3 **Research challenges** ............................................................................... 65

**Bibliography** ................................................................................................. 66

**A Publications** .............................................................................................. 71

**IEEE CLOUD 2019 Paper** ........................................................................ 72

**TPDS 2018 Paper** ....................................................................................... 77

**SAM 2018 Paper** .......................................................................................... 94
List of Figures

Figure 2.1: The Adaptation component and the Run-time model within the conceptual end-to-end architecture for data protection, as described in D3.3. ................................................................. 11

Figure 2.2: Overview of our approach. Boxes represent artefacts, ovals represent activities. ........ 13

Figure 2.3: Overview of the activities performed in the Adaptation component, as a Data Flow Diagram. The activities are also mapped on the phases of the MAPE model. .......................... 15

Figure 3.1: The structure of the proposed meta-model ................................................................. 17

Figure 3.2: The sub-model Data and its relations to the other sub-models ................................. 18

Figure 3.3: The sub-model Applications and its relations to the other sub-models ..................... 19

Figure 3.4: The sub-model Middleware and its relations to the other sub-models ...................... 19

Figure 3.5: The sub-model Infrastructure and its relations to the other sub-models ................... 20

Figure 3.6: The sub-model Actors. The dashed boxes indicate that all contained classes inherit from the same superclass ................................................................. 20

Figure 3.7: The sub-model Goals & metrics ................................................................. 21

Figure 3.8: The sub-model Mechanisms ................................................................. 21

Figure 3.9: TOSCA inheritance tree with additions. (Relations were omitted for clarity.) .......... 23

Figure 3.10: Example run-time model ............................................................................. 24

Figure 4.1: Risk pattern language ............................................................................. 25

Figure 4.2: Example risk patterns ............................................................................. 26

Figure 4.3: Overview of our implementation of the risk pattern approach ......................... 27

Figure 4.4: Realization of Risk Pattern A from Figure 4.2 as Henshin rule ................. 28

Figure 4.5: Realization of Risk Pattern B from Figure 4.2 as Henshin rule ...................... 29

Figure 4.6: Duration of pattern matching for growing cloud system models ................. 30

Figure 4.7: Adaptation algorithm ............................................................................. 31

Figure 5.1: Examples of the impact of VM selection decisions on the possibilities of VM placement ..... 33
Figure 5.2: Problem model using UML notation ........................................ 36
Figure 5.3: Minimization objectives of VM selection and placement ................. 40
Figure 5.4: Schematic example of applications deployed in a hybrid cloud ............... 49
Figure 5.5: Deployment of the CoCoME application ..................................... 58
Figure 5.6: Modified deployment of the CoCoME application after deploying another application. The numbers show the order in which the components are moved by the algorithm. .......... 59
Figure 5.7: Results of first adding, then changing, and finally removing 10 applications .................................................. 60
Figure 5.8: Different trade-offs between solution costs and execution time .................. 61
List of Tables

Table 5.1: Overview of used notation ................................................................. 37
Table 5.2: Results for the base setup (component sizes only) .................................................. 44
Table 5.3: Number of constraint violations plus number of overloads for different colocation constraints ............................................................. 45
Table 5.4: Notation overview ........................................................................ 50
Table 5.5: Components of CoCoME case study ................................................ 56
Table 5.6: Links and data transfers in CoCoME case study ........................................ 57
Table 5.7: Gain values of the movable components during the run of Algorithm 8 ................................................................. 57
1 Introduction

This deliverable presents the results of Work Package WP5, “Run-time data protection assurance”, achieved until month 34 of the project. In terms of the overall RestAssured architecture defined in deliverable D3.3, this deliverable details the Adaptation component and the Run-time model.

The overall goal of WP5 is to deliver novel monitoring and adaptation solutions for detecting and mitigating data protection violation risks in the cloud. To this end, WP5 pursues the following objectives and is accordingly structured into the following tasks:

- Task 5.1: Detecting privacy policy violations
- Task 5.2: Restoring privacy policy compliance by means of adaptations
- Task 5.3: Models@run.time as a shared knowledge base

In accordance with the project plan laid out in the DoA, we concentrated in the first half of the project more on task T5.1, and in the second half of the project more on task T5.2. Task T5.3 was processed continuously, since it provides a common basis for the other two tasks. Accordingly, deliverable D5.1 showed primarily results on tasks T5.1 and T5.3, while deliverable D5.2 laid the focus on task T5.2. In this deliverable, we combine all these results and present an integrated approach to run-time data protection assurance.

WP5 uses the concept of risk patterns (introduced in deliverable D5.1) for both detecting data protection violations and devising adaptations for mitigating data protection violations. In this context, there is a close interplay with run-time risk assessment developed in WP7 (described in deliverables D7.1 through D7.3). The run-time adaptation engine of WP5 considers also adaptations for goals other than data protection (e.g., to minimize costs); however, such adaptations are carried out only if the resulting configuration is deemed acceptable by the run-time risk assessment of WP7.

Adaptations elaborated in WP5 complement the work in WP4 and WP6 on data access protection according to data protection policies. While changes in individual data subjects’ data protection preferences are handled by policy changes (WP6) and by the effect of policy changes on data access (WP4), changes that affect the behavior of a whole system (e.g., because of changes in the environment) are covered by adaptations (WP5).

This deliverable is organized as follows. Chapter 2 gives an overview of the approach created in WP5 for run-time data protection assurance. Chapter 3 explains the used models that form the basis for both the detection and the mitigation of data protection violation risks. This is followed by the description of the methods for detecting and mitigating data protection violation risks in Chapter 4. Chapter 5 describes possibilities for including data protection awareness in optimization-based cloud adaptation approaches. Finally, Chapter 6 presents an outlook on future possibilities to extend the developed methods to the broader scope of fog computing.
2 Overview

This chapter gives an overview about the approaches developed in WP5. First we show how this work is embedded in the larger project context. Then we explain our overall approach spanning design-time and run-time activities. Finally, we show the interplay of the different run-time activities.

2.1 Embedding in the RestAssured architecture

Within the RestAssured run-time system, Adaptation is concerned with ensuring the continued satisfaction of data protection and other requirements despite changes. To this end, the managed systems (e.g., cloud infrastructure or applications) are monitored and a run-time model is updated so that it always mirrors the current state of the managed systems. The run-time model is searched for risks of data protection violation. If an intolerably high risk is identified, an adaptation is performed to mitigate the risk. This deliverable describes the conceptual operation of the Adaptation component and the Run-time model, which are important elements of the RestAssured run-time system (see Figure 2.1).

Figure 2.1: The Adaptation component and the Run-time model within the conceptual end-to-end architecture for data protection, as described in D3.3.

Searching for risks is done on two levels. On the first level, the Adaptation component itself performs a quick search in the Run-time model using the concept of risk patterns introduced in D5.1. On the second level, the Run-time risk assessment component of WP7 performs a more in-depth analysis using the techniques described in D7.1–D7.3. Through this separation, the Adaptation component by itself can figure out an adaptation that will likely lead to a good configuration in terms of data protection risks. The adaptation proposal has to be approved by the Run-time risk assessment component before the adaptation is actually executed. The two-level analysis of risks allows us to reduce the communication between the Adaptation component and the Run-time risk assessment component to the necessary minimum, thus significantly improving the performance of the overall RestAssured run-time system.

---

In this document, the term “risk” always refers to a risk of data protection violation.
2.2 From design time to run time

The compelling advantages of cloud computing, such as the seemingly infinite resource provisioning without the need for buying costly IT equipment, have made the cloud the platform of choice in many domains [11]. However, using the cloud is often associated with a loss of control since multiple parties – e.g., operators of cloud services – may potentially have access to data and code of applications in the cloud. Thus, storing and processing sensitive data in the cloud poses additional risks, hampering the adoption of the cloud [62].

There are several well-known techniques to ensure confidentiality and integrity of data in a cloud environment, e.g., encryption, signatures and other cryptographic techniques or authentication. However, they all have some drawbacks, e.g., in terms of performance overhead or inconvenience for users. Therefore, deciding which mechanism(s) to use is not trivial.

Traditionally, this decision was made at design time by experienced software engineers. However, to support agile deployment processes in a DevOps environment and to leverage the potential of the cloud in terms of flexibility, decisions on data protection mechanisms to use have to be made increasingly at deployment time or at run time. For example, an application may need only light-weight data protection mechanisms if deployed in a private cloud, but full-fledged data protection if deployed in a public cloud. During design time, it may not be clear where the application will be deployed [10, 66], so the application should be engineered such that the data protection mechanisms for the public cloud scenario are available but can be turned off if the application ends up being deployed in a private cloud. The mechanisms will then be turned on or off at deployment time. Moreover, the application may be migrated during run time between clouds using live migration [38], in which case data protection mechanisms may need to be turned on or off even during run time.

To enable this flexibility, decisions on the use of data protection mechanisms must be made automatically during deployment and run time, of course within bounds set during design time. Thus, the task of software and service engineers changes from making a concrete design decision to providing the decision logic and determining the information on which the decision should be based. In particular, this calls for run-time risk identification and assessment that can be used to trigger the appropriate mitigation actions during run time.

An important question is what information is needed to identify risks of data protection violations. We argue that this involves the following pieces of information in order to determine whether data protection policies may be violated:

- A model of the – current or planned – configuration of the relevant assets, including infrastructure elements, middleware, applications, and data, but also involved actors;
- A set of risk patterns, which describe asset configurations that would cause too high risks of data protection violation and hence must be avoided.

To address the challenges of ensuring data protection for dynamically deployed and configured cloud services, we propose a model-based approach spanning activities from design time to run time. Fig. 2.2 shows an overview of our approach.

Our approach revolves around two types of artefacts: Run-time model and risk patterns. The run-time model is created at deployment time and kept updated during run time, so that it always reflects the current configuration of the cloud-based application and its environment. The risk patterns are created at design time to capture configurations that must be avoided because of the associated high risk of data protection violations. These two artefacts are used by the “Risk identification” activity during deployment time and run time to identify risks of data protection violation.

The run-time model (also called model@run.time or MRT) is a collection of knowledge items about the cloud system, its environment, and its goals, that is made available at run time in a machine-readable format. In particular, it contains a model of the relevant technical cloud components of the infrastructure, middleware, and applications, as well as of the pieces of stored data and the relevant actors. Moreover, it contains
Risk patterns are the core concept in our approach for identifying potential data protection violations. A risk pattern describes a configuration of assets and actors that would lead to unacceptably high risk of data protection violations and hence must be avoided. Syntactically, risk patterns are expressed in terms of entities, attributes, and relations from the meta-model of the run-time model.

**Design time.** At design time, the meta-model of the run-time model is defined. This also determines the constructs that can be used to define the risk patterns.

Risk patterns are derived from a risk assessment. The risk patterns are designed as a black-list: Each unwanted configuration must be captured by a risk pattern. For determining risk patterns, a risk analysis process should be followed, focusing on the types of assets, vulnerabilities and threats in the system. Risks are identified and assessed in terms of their probability and impact. There are well-known methodologies and standards for such a process [18, 43]. More details can be found in deliverables D7.1–D7.3.

**Deployment time.** When the system is deployed in the cloud, the run-time model is created based on the target environment and the planned configuration of the application. When the run-time model is in place, the risk patterns can be evaluated for the first time to check whether the configuration would lead to any data protection issues. The evaluation of the risk patterns can be cast as a graph pattern matching problem, in which one tries to find a subgraph of the run-time model that matches the risk pattern. For this, graph pattern matching algorithms can be used [13, 37].

If a match was found, i.e., a risk was identified, the configuration is changed and checked again until a safe configuration is found.

**Run time.** During run time, the system can use self-adaptation to react to changes in its environment using the MAPE (monitor – analyze – plan – execute) model [30]. Monitoring is used to detect relevant changes and update the run-time model accordingly. For this purpose, existing cloud monitoring tools and further instrumentation can be used. The impact of observed changes is analyzed: the same evaluation logic as during deployment can be used to detect risk pattern matchings in the run-time model. If necessary, the same configuration logic as during deployment can be used to devise a new, safe configuration of the system (planning). Finally, the changes are executed by reconfiguring the system accordingly. The reconfiguration can be done either automatically or using operator-in-the-loop adaptation [25].

It should be noted that adaptations can be also triggered by other reasons (e.g., insufficient performance). Then, the pattern-matching logic is used to ensure that the new configuration of the system fulfills the data protection requirements.

---

**Figure 2.2: Overview of our approach. Boxes represent artefacts, ovals represent activities.**

---

D5.3 – October 31, 2019 13 of 112
2.3 The run-time activities

There are two main objectives for the Adaptation component. The primary objective is to mitigate high risks. The secondary objective is to improve the satisfaction of other functional and non-functional requirements (e.g., increasing efficiency, decreasing costs) of the managed systems. Since there can be multiple different adaptation possibilities in a given situation, we aim for choosing the “best” adaptation. Several criteria can be used to evaluate the different adaptation possibilities and to find the best one:

- The set of functionalities available to users in the target configuration
- Non-functional properties of the target configuration (e.g., financial costs and performance)
- The impact of the change from the current to the target configuration (e.g., amount of transient overhead or transient performance drop)

The concept of the Adaptation component follows the general structure of the MAPE model. Our structure differs slightly from the general MAPE model because the analyze and plan phases overlap in our approach, as shown in Figure 2.3.

In the monitoring phase, the Adaptation component listens for changes in the managed systems. The configuration of every managed system is stored in the Run-time model. If a change is detected, the model is updated and the analysis phase starts by checking the model for risks. Planning is invoked if an unacceptably high risk has been found. To eliminate the risks, the Adaptation component generates several possible adaptations, which are immediately checked for risks by the Adaptation component. Those adaptations that – based on the quick internal risk checking – mitigate the risk, are evaluated in terms of their impact on other goals. Starting with the best adaptation (according to the impact on other goals), the generated adaptation proposals are sent to the Run-time risk assessment component, until there is one that is approved by Run-time risk assessment. With the best adaptation found, the execution phase starts. The chosen adaptation is enacted on the managed system.
Figure 2.3: Overview of the activities performed in the Adaptation component, as a Data Flow Diagram. The activities are also mapped on the phases of the MAPE model.
3 The use of models@run.time

To enable effective adaptation at run time, a run-time model, i.e., a model of the system and its environment available for reasoning at run time, is of central importance [1, 21]. Therefore, our aim is to devise a run-time model of the cloud, which is useful for detecting and mitigating data protection violations. More specifically, we focus on the key modeling concepts required to cover the main elements of a cloud system in a run-time model of the cloud, leading to a cloud meta-model for data protection. The challenge in devising the meta-model of the cloud is to determine a sufficient level of detail as well as the necessary scope of modeled entities. Data protection concerns relate to all layers of the cloud stack, including, for example, secure hardware capabilities, co-location of virtual machines of different tenants on the same server, encryption of the communication between application components, and data anonymization. Moreover, the actors (organizations and individuals) as well as their goals and relations may play an important role. The challenge, therefore, is to devise a holistic meta-model encompassing all relevant entities.

3.1 Meta-model

3.1.1 Design considerations for the meta-model

Based on the generic requirements from deliverable D3.1 and the analysis of related work (see D5.1), the cornerstones for an appropriate meta-model can be established as follows.

- Often, data protection violation is not confined to a specific cloud layer, but arises from the interplay of entities belonging to different cloud layers. For instance, a data record accessed by an application hosted by a virtual machine on a physical machine might constitute a data protection violation because it potentially allows the administrator of the physical machine to access the data. However, if the data are encrypted, or the application is protected by appropriate access control mechanisms, or the physical machine supports secure hardware enclaves, data protection can still be ensured [40]. Therefore, it is important to model entities on different cloud layers, including their attributes as well as the interactions among them.

- Beyond technical entities like physical and virtual machines, also actors (natural or legal persons) and their attributes and relationships need to be modeled. For example, if the data belonging to actor A can get accessed by actor B, this may or may not constitute a data protection violation depending on whether A trusts B or not.

- Existing cloud models lack concepts for modeling data. For reasoning about data protection, we need to add support for modeling data. This is not to be confused with “data modeling,” which is about modeling the logical concepts captured by the data. For our purposes, other attributes of data objects matter, like their sensitivity (e.g., personal data) and where they are stored and processed.

- UML provides sufficient expressiveness to model cloud systems, as shown by [8]. Although there are also alternatives, we stick to UML because of its wide-spread adoption and the available tool support. More specifically, we will use UML class diagrams for the meta-model and object diagrams for the run-time model of the cloud.

- Beyond the mere detection of data protection violations, the model should also support finding the right mitigation action. For this purpose, it is vital to also model the possible data protection mechanisms and their impact on security attributes. Moreover, there may be multiple mechanisms that can be used to achieve the same security goal; in this case it is useful to select the one that has the smallest impact on other goals like performance or costs. For supporting such decisions, it is important to also model the impact of the available security mechanisms on those other goals.
• To be useful, the cloud meta-model needs to mirror the used technologies. However, there are several different technologies used in cloud computing and the technologies are also subject to change. For example, some cloud systems use virtual machines, others use containers, and a combination of the two is also possible. Therefore it is not feasible to strive for a cloud meta-model that is generic enough to capture all possible technical cloud realizations and at the same time also detailed and specific enough to allow reasoning about data protection impacts of a given cloud configuration. Rather, we argue that the exact cloud meta-model has to be created during system design, taking into account the specific technologies that are foreseen for the given system. We support that process by identifying the sorts of entities that need to be modeled, resulting in a framework for the meta-model, and giving examples of the modeling of entities that play an important role in most cloud systems.

3.1.2 Initial meta-model

Based on the considerations of Sec. 3.1.1, we proposed a cloud meta-model for data protection in D5.1. This meta-model can be seen as an extension of the existing multi-layer models of cloud systems discussed in the literature. We extended those models with several concepts that are vital for data protection, like explicitly modeling data and actors.

3.1.3 Structure of the meta-model

Independently from the used technologies, our meta-model framework is structured into four high-level packages as shown in Fig. 3.1 and explained below:

**Assets**: configuration of the cloud, including all the physical and virtual entities that are important for data protection

**Actors**: stakeholders and their roles relevant for data protection in the cloud at run time

**Goals & metrics**: non-functional properties that the system should fulfill

**Mechanisms**: possibilities for adaptation, including structural changes and changes to specific attributes

Fig. 3.1 also highlights the most important relations between the packages. In particular, assets may be owned and/or – independently from that – accessed by actors. In terms of relations among actors, trust is of special importance. We use a white-list approach to trust, i.e., every trust relation must be explicitly established (e.g., by means of a contract). Also, trust relations can be limited to specific types of actions on specific data. Goals & metrics relate to some assets, e.g., by specifying a response time constraint on an
application. The mechanisms change some assets and impact the goals & metrics. For example, encrypting a piece of data changes an attribute of that data object and it has some given positive impact on confidentiality but negative impact on performance.

The Assets package is further subdivided into Infrastructure, Middleware, Applications, and Data. Traditionally, cloud models include only the first three of those, but we also included data because of its obvious importance to data protection. Data is the primary asset at risk, but the other layers are also important because they act as additional attack surface.

Altogether, the proposed meta-model consists of seven sub-models: the four packages within Assets, plus Actors, Goals & metrics, and Mechanisms. These sub-models are detailed next.

3.1.4 Contents of the sub-models

The structure of the meta-model presented above is technology-independent and hence it can be expected to remain stable for a wide range of cloud systems. In contrast, the contents of the sub-models may depend on specific technologies and hence may differ from system to system. For example, the contents of the Data sub-model may depend on whether structured, semi-structured, or non-structured data are used; the contents of the Infrastructure sub-model will be different depending on whether virtual machines or containers are used etc. Therefore in the following we show examples of what the contents of the sub-models may be. Still, we try to be generic enough so that these models likely apply to many different cloud systems with no or little modification and can be used as starting point for modeling other cloud systems as well.

The Data sub-model shown in Fig. 3.2 is based on a relational model (like in [52]) but abstracts from details that are not important for us (e.g., domains of attributes) and adds others that are important (e.g., relating to the storage of data). The smallest unit of data is “Attribute value,” corresponding to a cell in a relational table. Attributes (columns) and Records (rows) contain multiple Attribute values; a Data set (table) contains multiple Attributes and multiple Records. A Data set can either be stored or transferred, represented by the entities Stored data set and Data flow inheriting from Data set. A database consists of multiple stored data sets. A Stored data set can be stored either in a local Database associated with a specific application Component or using a database management system (DBMS) provided by the Middleware. Attribute Values, Attributes, Records and Data Sets are all considered “Data objects.” For each Data object there can be multiple Data protection requirements. Data objects may be accessed by different Actors in Data-specific roles. Encryption can be used to alter the security attributes of a Data object. Between Data
and Infrastructure, there is no direct connection, but there are indirect connections, e.g., through applications that use the data and reside on the infrastructure.

Figure 3.3: The sub-model Applications and its relations to the other sub-models

Applications (Fig. 3.3) consist of multiple Components that are linked by Connectors. The logical structure of an Application is defined by an Application template, from which the specific Application, Component, and Connector instances are derived and scaled as necessary. SaaS (Software as a Service) developers work with Application templates. Applications are managed by SaaS operators and used by SaaS users. Certain Goals may apply to an Application and Authentication mechanisms can be turned on or off for an Application. Each Component is deployed in a VM (Virtual Machine) and controlled by an Application server. A Connector may accommodate Data flows.

Figure 3.4: The sub-model Middleware and its relations to the other sub-models

The Middleware sub-model (Fig. 3.4) supports multi-tier web applications with Web server, Application server, and DBMS entities. Web servers and Application servers can be clustered into Web server pools and Application server pools, respectively, which can be associated to a Load balancer. All these are Middleware elements, which are created, operated, and used by PaaS (Platform as a Service) developers, PaaS operators, and PaaS users, respectively. Web servers and Application servers are hosted by VMs, Applica-
tion servers manage application Components, and DBMS store data sets. A Load balancer has an associated Performance goal and can be configured by an appropriate configuration mechanism.

Figure 3.5: The sub-model Infrastructure and its relations to the other sub-models

In the sub-model Infrastructure (Fig. 3.5), an IaaS (Infrastructure as a Service) cloud consists of multiple data centers (DC); each DC consists of multiple network nodes, connected by links. Nodes include storage, switches, routers, and physical machines (PM); each PM may host multiple virtual machines (VM). IaaS clouds, DCs, Nodes, Links, and VMs are considered Infrastructure elements. An IaaS cloud can be accessed through a Public or Private cloud interface. The former allows access to VMs only, while the latter allows access to all Infrastructure elements. Different Goals may apply to Infrastructure elements. Application Components as well as Web and Application servers are hosted on VMs. An IaaS user can access VMs by using a Public IaaS interface, whereas IaaS developers and operators have access to any Infrastructure element. VMs can be scaled using Vertical scaling and their mapping to PMs can be changed by Migration.

Figure 3.6: The sub-model Actors. The dashed boxes indicate that all contained classes inherit from the same superclass

D5.3 – October 31, 2019 20 of 112
In the sub-model Actors (Fig. 3.6), each Party can have multiple cloud-specific or data-specific Roles. As Cloud-specific roles we differentiate users, developers and operators on SaaS, PaaS, or IaaS level. Within Data-specific roles, we differentiate Data subjects, Data producers, Data processors, and Data controllers, as stipulated by the GDPR [15]. Cloud-specific roles may relate to different Application, Middleware, and Infrastructure elements. Similarly, Data-specific roles relate to Data objects. A Party in a specific Role may have multiple Goals.

The sub-model Goals & metrics (Fig. 3.7) contains the non-functional requirements that the system must fulfill. This includes High performance, Low resource consumption, Data protection, User-friendliness, and Availability. These high-level goals can be decomposed into more specific goals; in particular, Data protection is decomposed into Confidentiality, Integrity, and Authenticity. Goals may relate to different Assets. In particular, Data protection goals relate to Data objects. A Goal is posed by a Party in a specific Role and may be impacted by different Mechanisms.

As shown in Fig. 3.8, Mechanisms can be of two kinds: Configuration or Structural change. Configuration
may mean that a feature such as Encryption within an application Component is switched on or off or some more subtle parameter change, e.g., the key length of the encryption algorithm is changed. Similar mechanisms are Authentication, Integrity protection (e.g., using cryptographic hashes), Anonymization of data, or Vertical scaling of VMs. Structural changes, on the other hand, create or remove entities or change interconnections among entities. In particular, Horizontal scaling of VMs creates or removes VM instances, whereas Migration of VMs between PMs changes the interconnection between the affected PMs and VM. These are powerful mechanisms to achieve several different Goals, including Data protection (e.g., by means of co-locating Components so that the Data flow between them does not have to traverse the network).

3.1.5 Updated meta-model

The initial meta-model described so far has proven useful in modeling different cloud systems and risk patterns. Several examples can be found in the attached SAM 2018 paper.

However, when working with the meta-model, two shortcomings became apparent. On the one hand, the elements used in the meta-model were lacking a detailed definition, which led to small inconsistencies and misunderstandings. On the other hand, a proprietary meta-model hinders interoperability. In D5.1 we showed that a mapping between our meta-model and the standard TOSCA modeling language is to some extent possible; however, it became also clear that creating and maintaining such a mapping is difficult and does not cover all of the used modeling constructs.

To address these shortcomings, we decided to change our meta-model so that it is aligned with TOSCA. The TOSCA modeling language is a standard, published in 2013 by the Organization for the Advancement of Structured Information Standards (OASIS). It is used for deployment and portability of cloud applications. Furthermore, as a standard it allows easier interoperability and reusability of components.

TOSCA consists of two parts, the topology and the management plans. For our purposes, however, only the topology is relevant, thus only this one is used here. There are two different types of elements in the topology: nodes and relations. The nodes represent the individual components, while the relations represent the connections between the nodes. The nodes and relations are represented as node templates and relation templates in a topology template. Several objects can be instantiate on the basis of a template.

The TOSCA standard requires the implementation of at least the basic (normative) TOSCA nodes and the corresponding relations. To represent all relevant aspects of a cloud system, more nodes may be needed. To find risky configurations within a cloud system model, it is crucial that the relevant actors are displayed in said model. The TOSCA standard, however, does not contain a default node type to represent actors. Thus an addition of several nodes aimed to represent the most important actors is made (see Figure 3.9). All these nodes inherit from the abstract node "actors". From there they can be grouped into one of two categories, "DataSpecificRole" or "CloudSpecificRole". In the "DataSpecificRole" category are all actors, that have immediate access to data, either because they produce, process, control, or own the data. On the other hand, in the "CloudSpecificRole" category the actors have an immediate relation to a cloud service, meaning they are either developer, operator or user of the cloud service. These roles are then divided into the tree different layers, IaaS, PaaS and SaaS.

Furthermore the data, which is the focus of the search for risks, has to be modeled as well. For this purpose, four more nodes are added. The abstract node "DataSet" from which the nodes "DataFlow" and "StoredDataSet" inherit, as well as the node "Record" which inherits directly from "tosca.nodes.Root". The "Record" node represents data entries, which are collected within a "DataSet". The difference between "StoredDataSet" and "DataFlow" is, that "StoredDataSet" is used to model a data set that is persisted within a database, while "DataFlow" represents data sets that are being transferred.

The implementation required a node that contains all other nodes. To this end the "CloudEnvironment" node is added.

The decision to use TOSCA as basis of the meta-model is influenced by the fact, that TOSCA is a standard.
that is also used in the industry.\footnote{\url{https://wiki.oasis-open.org/tosca/TOSCA-implementations}} Through the use of a standard it is easier to achieve interoperability. At first a proprietary meta-model was used, but the advantages of a meta-model based on a standard caused a switch to TOSCA as basis. The quick change from one meta-model to another was possible because the Adaptation component is generic enough so that it can work with different meta-models. Because of this design it is possible to modify the meta-model without changing the actual code of the Adaptation component.

3.2 Run-time model

A run-time model represents the data-protection-related configuration of a specific cloud system. The run-time model describes the system itself as well as its environment. The run-time model is based on the meta-model, with the meta-model being an UML class model and the run-time model being an instantiation in the form of an UML object model. The run-time model is created during deployment and whenever the configuration changes during run time, the run-time model gets updated to represent the current configuration.

The run-time model is a very important part of our approach, because it provides the basis for analyzing risks. The reasoning on risks is carried out based on this model.

In Figure 3.10 an example run-time model is shown, using UML object diagram notation. This example is taken from \cite{53}.

In the example, there is a “DataSubject” that has some data in form of a “Record”, which is part of a “StoredDataSet”. The “StoredDataSet” is stored in a “Database”, which is hosted on a “DBMS”, which is provided by a “PaaSOperator”. From the “Database” three different “SoftwareComponents” access the data. Additionally there are connections between the “DataSubject” and the other “Actors”. Those connections

\footnote{\url{https://wiki.oasis-open.org/tosca/TOSCA-implementations}}
show whether the "DataSubject" trusts the other "Actors".

Figure 3.10: Example run-time model
4 Detection and mitigation of data protection violation risks

In this chapter, we describe in more detail the main parts of the proposed approach that was presented in Section 2. In particular, we describe the risk patterns and how they are captured and applied with the help of a set of open-source software tools. We present empirical measurements on the scalability of the approach as well as the method for determining adaptations to mitigate identified data protection risks.

4.1 Risk patterns

Cloud systems represent a complex interplay of different components. Each of the components can be individually free of risks, but through the interplay with other components a risk may still arise. To combat this problem, our approach introduces risk patterns. Risk patterns are used to describe patterns within a run-time model. These patterns characterize configuration within a cloud system with unacceptably high risks of data protection violation. Thus any instance of a risk pattern within a run-time model is an unacceptably high risk in a cloud system and has to be mitigated.

The necessary pieces of information to create a risk pattern are a risky configuration that should be avoided, and a language to represent the risk pattern. An IT security expert is supposed to identify risky configurations within the cloud system and to abstract them into patterns. The pattern has to be modeled in a more general form than the run-time model but still in a way, that is compatible with said run-time model. This is the aim of the risk pattern language. The risk pattern language refers to the meta-model and uses a notation as shown in Figure 4.1.

The risk pattern language is based on the notation of UML object diagrams. A difference between the risk pattern language and the UML object diagram stems from differences in the semantics. In an UML object diagram, a node represents a specific object, while in the risk pattern language a node represents any instance of that type, fitting the description in the risk pattern. Furthermore, a node in a risk pattern may also match a node of a type that inherits from the type in the risk pattern.

In the risk pattern language, nodes are represented by a square with the name of the node inside. The attributes of a node are optionally listed in the lower section of a node square. The value of an attribute is set by writing the name of the attribute, an equal sign and finally the value of the attribute. This way a node with this attribute and the specified attribute value has to exist for a match to be found. If between the equal sign and the value there is an exclamation mark, the node with the attribute is not allowed to have this specific attribute value for a match to be found.

There are two different ways to display edges between nodes. The first one is a solid arrow. It symbolizes an edge that has to exist within the model for a match. The second way is a dashed line with an empty circle at the end. This edge represents a relation, that is not allowed to exist for a match. Furthermore, even abstract nodes can be used to generalize a pattern.

The nodes and edges fitting the description of a risk pattern, that are found within a run-time model,
Figure 4.2: Example risk patterns

Figure 4.2 shows two example risk patterns. These examples are taken from [53] and fitted to the current meta model and the risk pattern language. Risk pattern A represents a situation where a data subject stored their private but unencrypted data in a database which is also not encrypted. This database is operated by a PaaS operator. The relation between data subject and the PaaS operator is a non-trusting relationship. Thus the PaaS operator might misuse the data of the data subject. The non-trust relation between the data subject and the PaaS operator is visualized through the dashed line edge as can be seen in Figure 4.1.

The second risk pattern in Figure 4.2 (risk pattern B), represents a situation, where a data subject, which is a EU citizen stored their data in a database, which is hosted on a physical machine outside the EU. Through the EU General Data Protection Regulation (GDPR) this is not allowed. The fact that the machine is not to be inside the EU (because it would not violate the GDPR otherwise) is visualized through the use of an exclamation mark before the value of the location attribute of the “Compute” node.

Further examples of risk patterns can be found in the attached SAM 2018 paper.

4.2 Implementation of the risk pattern approach

For implementing the risk pattern approach, we need more than just a graph pattern matching algorithm. As can be seen from the previous section, we need the ability to reason about not only graph structures (vertices and edges), but also object types, attribute values, and relationship types. The types can also form an inheritance hierarchy. E.g., it was shown in [45] that it is useful to extend the meta-model with two new types AtomicComponent and CompoundComponent, both inheriting from Component. The Component element of Risk Pattern B in Figure 4.2 should then match any object of type Component or of any of its sub-types.

Therefore, we decided to use the Eclipse Modeling Framework (EMF [58]). EMF is a widely used modeling framework based on the Eclipse environment and supported by a variety of tools. In particular, Henshin is a model transformation library, also supporting pattern matching in EMF models [4]. EMF also allows direct model manipulation with custom Java code.

4https://gdpr-info.eu/
Figure 4.3: Overview of our implementation of the risk pattern approach

Figure 4.3 shows the interaction of the used tools (EMF, Henshin, Sirius, MatchFinder) and the artefacts created or used by the tools (meta-model, cloud system model, risk patterns). With the help of EMF, we created a meta-model of cloud configurations. This meta-model is stored in EMF’s native format (Ecore). The risk patterns are modelled with the Henshin editor in the form of Henshin rules. Pattern matching and the handling of found matches are performed by the MatchFinder tool, which is a combination of custom Java code and the Henshin engine. To foster experimentation, visualization, and manual exploration of design alternatives during design time, we also created an editor with which the cloud system model can be displayed and edited. For this purpose, the Sirius plug-in was used [63]. In the following, we describe each component of Figure 4.3 in more detail.

Meta-model in Ecore We translated the meta-model from [45] to Ecore format. This required some small technical changes, like the insertion of a root node. We also discovered some small inconsistencies between the meta-model of [45] and the cloud system model and risk patterns of [53]; e.g., the meta-model of [45] contained no relation between actors that would represent a trust relationship, which is needed in the example risk patterns of [53]. We fixed the inconsistencies by making appropriate changes to the meta-model, the cloud system model, or the risk patterns. Such inconsistencies can remain undiscovered in a manual validation such as in [53, 41, 45]; an implementation and validation like we perform here has a much higher chance of discovering such issues.

Risk patterns as Henshin rules Henshin provides an editor for the creation of Henshin rules based on an Ecore model. A Henshin rule specifies a model transformation. Conceptually, a Henshin rule consists of two parts: the left-hand side (LHS) specifies a pattern to be found in the input model, while the right-hand side (RHS) specifies how the found pattern is to be changed in the output model. The Henshin editor presents the LHS and RHS together in an integrated object model, in which Henshin-specific annotations are used to mark the role of the model elements (objects, relations, attributes) with respect to the LHS and RHS. For specifying risk patterns, we use two of Henshin’s annotations: preserve and forbid. Every model element annotated with preserve has to be found in the input model for a match. Model elements marked with forbid must not be present for a match to be found. Figures 4.4–4.5 show the realization of the two example risk patterns from Figure 4.2 as Henshin rules. As can be seen, the annotations preserve and forbid are used to mark model elements that must be present, respectively must not be present in a match.
Editor for cloud system models  
Beside the risk patterns, a cloud system model is needed in which the risk patterns are searched for. The cloud system model can be a design-time artefact that a cloud security analyst works with to analyze different cloud configurations, or a run-time model updated through monitoring reflecting the current configuration of the cloud system. We created an editor for working with cloud system models, which may be used in the design-time setting by the cloud security analyst. In a run-time setting, this tool may be used by a human operator to visualize the current system configuration.

We used Sirius [63] to create a graphical editor, in which objects, their attributes and relationships can be created in accordance with the meta-model, displayed on a canvas, and edited. We constructed creators for the instantiable types of the meta-model, which are made available to users in the form of a toolkit. For the relations, providing a separate creator for each association in the meta-model would have led to a huge number of creators, resulting in decreased usability. Hence we implemented a generic creator for relations, which automatically infers the type of the relation based on the types of the start and end nodes.

The control logic  
MatchFinder is a Java program. It uses the risk patterns in form of Henshin rules, the cloud system model created with the Sirius-based editor, and the meta-model in Ecore format. MatchFinder controls the pattern matching process and handles the results. The SearchInitiator within MatchFinder reads and converts all input models and initiates the pattern matching using the Henshin tool set. The result of the pattern matching is a (possibly empty) set of matches. For each match, MatchFinder displays a human-readable textual description of the match. The matches are also graphically shown within the cloud model in the Sirius-based editor.

More details on the implementation of the approach can be found in the IEEE CLOUD 2019 paper in the appendix.
4.3 Empirical results

We performed controlled experiments to investigate the scalability of our approach. To generate larger models with a realistic structure, we enlarged the example cloud model from Section 3.2 in a sequence of 10 steps. After each step we checked the duration needed by MatchFinder to search for (i) Risk Pattern A, (ii) Risk Pattern B, (iii) both risk patterns together. The enlarged models consist of several copies of the example model which share the same data subject node. The model of the first step consists of the basic example model with 18 nodes. Each further step adds the example model 100 times except the data subject node. Thus the model of the second step consists of $18 + 100 \cdot (18 - 1) = 1,718$ nodes and the model of the 10th step contains $18 + 10 \cdot 100 \cdot (18 - 1) = 17,018$ nodes. The measurements were performed on a Surface Pro 3 laptop with Core i5-4300U CPU at 2.900MHz, 4GB RAM, Windows 10 Professional, and Java 9.

Fig. 4.6 shows the duration of MatchFinder for the models of increasing size. As can be seen, the execution time scales roughly linearly with the size of the model. Looking for Risk Pattern A which consists of only 5 nodes needs less time than looking for Risk Pattern B which consists of 7 nodes. Looking for both risk patterns at the same time needs more time than looking for just one; however, if both risk patterns have to be looked for, then it is faster to look for them together than to look for them separately. This is probably due to the overhead of setting up the search itself, which is incurred only once if the two risk patterns are
searched for together.

Most importantly, Fig. 4.6 shows that the duration is quite low even for the biggest models tested. In the first step, the search took 104 ms while the duration in the last step is 346 ms for a model with over 17,000 objects. Thus the duration increases by around 3.3 times while the size of the model increases by around 945 times. These findings indicate that the presented implementation scales well even for big cloud systems.

We also investigated how execution time scales with the number of edges in the model. We took one of the models from the previous experiments and added random edges to it, thus keeping the number of nodes constant while increasing the number of edges. We repeated this experiment with several starting models of different size. Also, we considered two approaches for adding new edges: (i) only adding edges inside the copies of the original example model of 18 nodes, (ii) adding edges between copies of the original example model. Our experience from these experiments is that increasing the number of edges hardly influences the time needed for the pattern matching. The duration remains low even for the biggest models considered (more than 100,000 edges).

4.4 Mitigation of data protection violation risks

The risky configurations that are identified have to be mitigated to ensure the protection of the data. To this end, our approach uses adaptations. Adaptations have the advantage that they can be executed automatically during run time.

When one or more risky configurations within a run-time model are identified, the model has to be adapted to mitigate the risks. For this purpose, when the risk patterns are defined, possible adaptations to mitigate those risks are captured as well, in the form of model transformation rules.

To cope with the identified risks, possibly multiple adaptations are needed. To find the appropriate combination of adaptations, our algorithm creates a search tree. The root node represents the current run-time model. Within the run-time model there are risky configurations identified through the use of the risk patterns. The child nodes of a node represent the run-time models obtained from the parent node by an adaptation. Thus for every adaptation that belongs to an instance of a risk pattern found in the run-time model, there is a child node. This leads to a large, potentially infinite tree. Obviously an infinite tree cannot be searched completely. Thus an algorithm to search for adaptations that mitigate all found risks is needed. In this approach the search algorithm “best first search” is used. In this algorithm, a heuristic is
used to determine how promising a node of the search tree is, i.e., how likely it is that the descendants of the node will lead to a risk-free node quickly. Based on this heuristic, the algorithm always selects the most promising node for further exploration. If there is a tie, the next node to explore is selected randomly among the equally promising nodes. To estimate how promising a node is, the heuristic may use for example the number of risks found within the run-time model of that node. With fewer risks within the run-time model, it is more likely to mitigate the found risks with as few as possible adaptations. If a node has no more risks, then the path consisting of the adaptations used, to reach this node from the root node is saved as a possible solution.

After a predefined amount of time, the found solutions are compared. The comparison is done based on the quality of the solution models in terms of functional and non-functional properties. That is, the amount of functionality available to users is evaluated, as well as non-functional properties such as financial costs. The best solution should have a high rate of preserved functionalities and a low rate of costs. The best solution is then selected and the corresponding adaptations are executed on the cloud system.
5 Optimization-based approaches

So far, we presented our general approach for detecting and mitigating data protection risks, in which other goals (minimizing costs, maximizing performance etc.) played an important but subordinate role. In this chapter, we take a complementary view, focusing on how data protection requirements can be enforced in the course of optimization of other system goals. We present two such approaches. The first is on handling colocation and anti-colocation constraints (e.g., to ensure that software components handling personal data are not colocated with potentially malicious software components) in cross-layer cloud resource optimization. The second presented approach deals with the optimized deployment of applications in hybrid clouds, ensuring that components dealing with sensitive data remain in the private cloud.

5.1 Handling colocation risks in cross-layer cloud optimization

As cloud data centers (DCs) are serving an ever growing demand for computation, storage, and networking, their efficient operation has become a high priority. Cloud providers seek to serve as many customer requests as possible and to decrease operational costs. Operational costs are largely driven by electricity consumption, which also impacts the environment. At the same time, cloud providers must also fulfill service-level objectives (SLOs) on performance, availability, and security.

Virtualization has been widely adopted in DCs to consolidate workload on the necessary number of physical machines (PMs) with high utilization of the available hardware resources. For this purpose, virtual machines (VMs) are used as the virtual infrastructure for running the workload, enabling the isolated execution of multiple applications on the same PM. However, virtualization also has some drawbacks (e.g., overhead [73]) and limitations (e.g., no perfect isolation of colocated VMs from each other [9, 34]).

Because of its impact on costs, application performance, SLOs, and the environment, optimization relating to the management of VMs has received considerable attention in the last couple of years. As shown in our recent survey [36], most previous research efforts fall into one of two categories: VM placement and VM selection. VM placement is a problem faced by Infrastructure-as-a-Service (IaaS) providers: how to determine a mapping of VMs to PMs with the main objective of minimizing overall energy consumption. On the other hand, VM selection is faced by IaaS tenants concerned with assigning application components to VMs.

The two problems are quite different: VM placement is about physical resources, their utilization and power consumption, whereas VM selection is concerned with lease costs and application-level performance metrics. The central notion that connects the two perspectives is the VM.

Although VMs play an important role, especially in a public IaaS setting, we argue that VMs are just a tool for mapping tenants’ application components to the provider’s PMs in a safe and manageable fashion. Tenants’ main objective is to find hosts for their applications, providers’ objective is to utilize their infrastructure by accommodating workload that is valuable for their clients, and thus realize revenue. VMs can be seen as wrappers around application components that make all this possible in a manageable way. In this respect, VM placement and VM selection are just two sides of the same coin. Most importantly, the two problems influence each other.

A simplified example is shown in Fig. 5.1. Here, we consider a single resource dimension (e.g., only CPU) and assume that all PMs have the same capacity according to this resource. The capacity of the PMs is taken to be 1. We consider six components with resource need 0.3 each (i.e., each component requires 30% of the capacity of a PM). Further, we assume that a VM adds an overhead of 0.05 to the size of the contained component(s) in terms of resource consumption. The three subfigures show the effect of different VM selection policies on VM placement. In Fig. 5.1(a) the VM selection policy selects a dedicated VM for each component, resulting in 6 VMs of size 0.35 each, the placement of which requires at least 3 PMs. In

1The term “component” denotes a component of an application, i.e., a software component.
Fig. 5.1: Examples of the impact of VM selection decisions on the possibilities of VM placement

Fig. 5.1(b) components are grouped pairwise into VMs, resulting in 3 VMs of size 0.65 each, the placement of which again requires 3 PMs. In Fig. 5.1(c), groups of 3 components are mapped to VMs, resulting in 2 VMs of size 0.95 each, and these can be hosted by 2 PMs. Therefore, this third scenario leads to approximately 33% energy savings. However, if we continue this line of thought and map 4 components into a single VM, this would result in VMs of size 1.25, which cannot be accommodated by the available PMs without severe resource overload.

As demonstrated by this example, VM selection influences VM placement in a non-trivial way. Therefore we argue that, at least in a private cloud setting, where VM selection and VM placement are in the hands of the same organization, the two kinds of optimization should be carried out in a closely coupled way. We consider the IT department of an organization running a private cloud, in which both VM selection and VM placement are carried out by the IT department. The main questions we address are:

- How much can we gain by optimizing VM selection and VM placement together, in a joint optimization?
- If the two problems are solved separately, how much can we gain by incorporating knowledge about VM placement into VM selection and vice versa, by incorporating knowledge about VM selection into VM placement?

To answer these questions, we compare several algorithms, from an integrated approach for the joint selection-and-placement problem to complete separation of the two problems. Also some approaches are investigated that are in between, meaning that they solve the two problems separately but include some information about one of the problems into the solution of the other. To compare the algorithms, we use several metrics including energy consumption, license costs, compliance with hardware affinity constraints, and compliance with colocation constraints of the resulting system configuration.

5.1.1 Aspects of the component–VM–PM mapping

Depending on the characteristics of the application components, the used VM selection and placement strategies, and general DC management policies, there can be many different aspects that must be taken into account in the trilateral component–VM–PM mapping. In the following, we provide an analysis of the most important aspects, grouped into three categories: aspects relating to VM sizing, co-location, and other affinity aspects.

5.1.1.1 VM sizing

The following aspects of the trilateral component–VM–PM mapping must be taken into account in relation with the sizing of VMs:
• **Placeability.** As demonstrated by the example in Fig. 5.1, the way components are mapped to VMs influences the size of the resulting VMs, which in turn determines the possible placements of the VMs on the PMs and hence the costs of the placement. It is difficult to make good decisions in VM selection because the placeability of a VM depends not only on its own size, but also on the size of other – existing and future – VMs.

• **Possibilities of live migration.** VMs provide a generic and transparent facility for migration in order to adapt to changes in the workload. Although some applications may offer the possibility to move individual components from one VM to another, this cannot be assumed in general. Hence we assume that only VMs can be migrated. The VMs are thus the unit of migration, which means that the VMs’ sizes determine the granularity of migrations [39]. From this point of view, it is beneficial to have small VMs because they allow a fine-granular control of the PMs’ utilization, thus avoiding fragmentation and achieving near-optimal utilization.

• **Overhead.** Virtualization introduces some overhead in terms of resource consumption. Since every VM adds some overhead (e.g., the size of the guest operating system), from this point of view it is beneficial to have a lower number of larger VMs. Obviously, this contradicts the above aspect which would lead to many small VMs, so that a good balance has to be found between the two aspects.

5.1.1.2 **Co-location**

The following aspects all lead to some kind of constraints or preferences on co-location of components in the same VM and/or PM:

• **Communication.** For components that communicate with each other, it may be necessary or at least advantageous to map them to the same VM, or at least to VMs on the same PM [50]. For example, a legacy application (i.e., one that was not developed with a multi-VM deployment in mind) may consist of multiple components that communicate via shared memory; in this case, these components have to be mapped to the same VM. If they communicate for instance through TCP sockets, then such a colocation is not necessary, but for communication-intensive applications it may still be advantageous in order to reduce latency and save network bandwidth.

• **Data protection.** Although virtualization provides a level of isolation between co-located VMs, it does not provide sufficient defense against malicious attacks. Using covert channels in hardware or software, a malicious VM can gain sensitive information from co-located VMs [33]. Therefore it is important to co-locate VMs hosting critical components only with VMs hosting only trusted components. In a private cloud, data protection concerns are typically lower than in public clouds, but in critical domains (e.g., banking), it is still important to isolate critical productive components from the ones whose security is not guaranteed.

• **Fault tolerance.** For some components, high availability may be necessary to improve reliability [72]. This has two important consequences for selection and placement. First, such components should not share a VM with other, less stable components to avoid that the failure of another component crashes the VM. (In contrast, they may be on the same PM since a VM can tolerate the crashing of a co-located VM.) Second, if a component is replicated to guarantee high availability, then the replicated instances should be placed on different PMs so that a PM fault impacts only a single instance.

• **Performance interference.** Components in the same VM compete directly for all system resources. Between VMs, the virtualization layer partitions some resources (e.g., memory space), but this isolation is far from perfect, so that for some resources – like memory bandwidth or caches – there is also competition between co-located VMs, which may lead to significant performance degradation [29]. Therefore, components that use the same resource intensively, should be packed into distinct VMs, possibly also on distinct PMs.
• **Correlated load peaks.** Sudden load increases are dangerous in servers with high utilization because resource overloads may easily lead to SLO violations. It is especially problematic if the load of multiple components on the same PM or VM increases at the same time. Therefore, co-location of several correlated components should be avoided [35].

5.1.1.3 Other affinity aspects

The following aspects, leading to software or hardware affinity preferences, must also be taken into account when mapping components to VMs and VMs to PMs:

- **Operating system dependency.** In many cases, application components depend on a specific operating system (OS), a specific version (or range of versions) of an OS, or are compatible only with a set of OSs. While the OS of the VM can be chosen independently from the host OS, the OS of the VM must match the OS requirements of the components.

- **Hardware affinity.** Some components may require some special hardware feature, or there may be a preference for such. For example, a component may run only on a PM with a GPU of a given vendor, or it may be able to take advantage of such hardware to boost its performance.

- **Licensing.** Some components may require costly licenses. There are many licensing models, several of which are agnostic of placement: e.g., if license cost depends on the number of users, then the placement of the component is irrelevant. However, some licensing constructs specify fees depending on the number of machines (either VMs or PMs) on which the software runs, in some cases also weighting the number of machines with some coefficients of computing power. For such licensing models, the placement of the components does matter: colocating multiple components with the same license on the same VM or PM leads to a reduction of the license fees to be paid.

5.1.1.4 A note on component sizing

Our focus is on software deployment. At this stage we can assume that the components and their sizes (i.e., resource requirements) are given. The components and their sizes must be determined in an earlier phase of the software development process in such a way that the performance of the application complies with the SLOs that it has to fulfill.

5.1.2 Problem formulation

Based on the analysis of the relevant aspects of the problem, we came to the problem model summarized in Figure 5.2.

The problem model revolves around components that are deployed in VMs, which in turn are deployed in PMs. Let \( C, V, \) and \( P \) denote the set of components, VMs, and PMs, respectively. For a component \( c \in C \), \( v(c) \) denotes the VM where \( c \) is deployed; likewise for a VM \( v \in V \), \( p(v) \) denotes its hosting PM.

The size of a component encodes its resource requirements along multiple resource types as a \( d \)-dimensional vector. Here, \( d \) is the number of considered resource types, e.g., if CPU and memory are considered, then \( d = 2 \). The size of a VM is also a \( d \)-dimensional vector: the sum of the sizes of the components deployed in the given VM, plus the overhead of virtualization (the size vector of an empty VM). For a VM \( v \in V \), its size thus is computed as

\[
\text{size}(v) = s_0 + \sum_{c \in C: v(c) = v} \text{size}(c),
\]

\(^2\)Graphical Processing Unit

\(^3\)The numbers near the ends of links mean minimum and maximum cardinalities and * means infinity. For instance, the numbers near the link between Component and VM mean that a component is deployed in exactly one VM, whereas a VM can host an arbitrary number (from 0 to infinity) of components.
where \( s_0 \in \mathbb{R}_+^d \) is the size vector of an empty VM.

Each PM \( p \in P \) has given capacity according to each of the considered resource types. Therefore, the capacity of a PM \( p \) is given by a \( d \)-dimensional vector \( \text{cap}(p) \). The mapping of VMs on PMs must respect the capacity of the PMs, as captured by the following constraint:

\[
\forall p \in P : \text{load}(p) = \sum_{v \in V : p(v) = p} \text{size}(v) \leq \text{cap}(p).
\]

Note that here, “\( \leq \)” is a component-wise comparison of \( d \)-dimensional vectors: for \( x, y \in \mathbb{R}^d \), \( x \leq y \) if and only if \( x_j \leq y_j \) for each \( j = 1, \ldots, d \).

The state of a PM can be either on or off. The operating system of VM \( v \) is \( \text{os}(v) \). For each component \( c \), the list of operating systems is given on which it can run; \( \text{os}(v(c)) \) must be an element of this list.

A colocation constraint relates to a pair of components. The type of the constraint can be one of must, should, should not, and must not. Moreover, it is given for each colocation constraint whether it relates to the colocation in the same VM or the same PM. With this mechanism, we can model all colocation aspects described in the online supplemental material. For instance, shared-memory communication between components leads to a must-constraint on VM level, meaning that they must be in the same VM, whereas intensive but loosely-coupled communication may lead to a should-constraint on PM level, meaning that they should be in the same PM. Data protection concerns may necessitate a must-not-constraint on PM level, meaning that they must not be in the same PM etc.

A component may have a license assigned to it, if it is placement-relevant because the license fee is proportional to either the number of VMs or the number of PMs running components with the given license. For a VM-based license \( \ell \), let \( V(\ell) \) denote the set of VMs containing at least one component associated with license \( \ell \), then the license fee to be paid because of \( \ell \) is \( \text{fee}(\ell) \cdot |V(\ell)| \). Similarly, if \( \ell \) is a PM-based license and \( P(\ell) \) denotes the set of PMs containing at least one component associated with \( \ell \), then the license fee to be paid because of \( \ell \) is \( \text{fee}(\ell) \cdot |P(\ell)| \). The total license fee to be paid is the sum of the fees for each license.

As a consequence, if multiple VMs containing components with the same license \( \ell \) are in the same PM, then

- the license fee has to be paid only once if \( \ell \) is a PM-based license;
- it has to be paid for each VM if \( \ell \) is a VM-based license.
Table 5.1: Overview of used notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>Set of all components</td>
</tr>
<tr>
<td>$V$</td>
<td>Set of all VMs</td>
</tr>
<tr>
<td>$P$</td>
<td>Set of all PMs</td>
</tr>
<tr>
<td>$v(c)$</td>
<td>VM hosting component $c$</td>
</tr>
<tr>
<td>$p(v)$</td>
<td>PM hosting VM $v$</td>
</tr>
<tr>
<td>$d$</td>
<td>Number of considered resource types</td>
</tr>
<tr>
<td>$s_0$</td>
<td>Size vector of an empty VM</td>
</tr>
<tr>
<td>$V(\ell)$</td>
<td>Set of VMs with at least one component associated with license $\ell$</td>
</tr>
<tr>
<td>$P(\ell)$</td>
<td>Set of PMs with at least one component associated with license $\ell$</td>
</tr>
<tr>
<td>$W(x)$</td>
<td>Power consumption of a PM with CPU load $x$</td>
</tr>
<tr>
<td>$W_{\text{min}}$</td>
<td>Minimum power consumption of a PM</td>
</tr>
<tr>
<td>$W_{\text{max}}$</td>
<td>Maximum power consumption of a PM</td>
</tr>
</tbody>
</table>

A PM may possess some PM features. A hardware (HW) affinity constraint can specify the relation of a component to a PM feature. The type of the HW affinity can be either must (the component definitely requires a PM with the given feature) or should (the component benefits from a PM with the given feature).

The power consumption of a PM is a function of its CPU load. As in several previous works [6, 20, 60], we use a linear approximation, i.e., the power consumption of a PM with CPU capacity $c$ and CPU load $x$ is given by

$$W(x) = W_{\text{min}} + (W_{\text{max}} - W_{\text{min}}) \cdot x/c,$$

where $W_{\text{min}}$ and $W_{\text{max}}$ are the minimum and maximum power consumption of the PM, respectively. Table 5.1 gives an overview of the used notation.

Now we summarize the problem’s inputs, outputs, constraints, and objectives. The inputs are:

- the set of components with the associated colocation constraints, licenses, and HW affinities;
- the set of PMs with their PM features.

The output consists of:

- the set of VMs to be used,
- the mapping of components to VMs,
- the mapping of VMs to PMs.

The solution must respect the PM capacity constraints, the requirements of the components in terms of VM OS, the colocation constraints of type “must” and “must not,” and the HW affinities of type “must.” There are multiple objectives: minimizing the total energy consumption of the PMs, minimizing the total license fee, maximizing the number of satisfied colocation constraints of type “should” and “should not,” and maximizing the number of satisfied HW affinities of type “should.”

As with any model, this problem formulation also abstracts from some technical details. For example, the transient processes of turning a PM on or off, deploying a VM on a PM, or deploying a component in a VM are not considered, nor their overhead in terms of time and energy. This is in line with the problem formulations used in most previous works in this area [60] and represents a good approximation for long-running software components. For very dynamic settings, the problem formulation – and the subsequent algorithms – may need to be extended in this respect.

5.1.3 Mapping algorithms

In this section, we first devise an algorithm for the joint VM selection and VM placement problem, called COMBINED. Then, for the purpose of comparison, we also introduce some algorithms for only VM selection respectively only VM placement.
VM placement and VM selection are tough combinatorial problems for which optimal methods are unfortunately intractable for practical problem sizes [38]. Therefore, in line with most existing works, the algorithms presented here are all heuristics.

5.1.3.1 Combined VM selection and VM placement

The aim of the COMBINED algorithm is to map a new component $c$ on a VM and a PM. This can be done in several ways:

- Starting a new VM $v$ to host $c$ and placing $v$ on a PM
- Selecting an existing VM $v$ to host $c$ and keeping $v$ on the PM where it is
- Selecting an existing VM $v$ to host $c$ and migrating $v$ to another PM

These three principal ways of mapping can be unified by considering all VMs from $V \cup \{v^*\}$ as possible hosts for $c$, where $V$ is the set of existing VMs and $v^*$ is a new VM, and considering all PMs as possible hosts for the selected VM. The basic idea of the COMBINED algorithm is to examine all these possibilities and choose the best one.

One challenge that needs to be tackled is the potentially large number of possible configurations to examine, namely $(|V| + 1) \cdot |P|$. The naive approach of examining all possible configurations can be rather time-consuming if $|V|$ and $|P|$ are large, taking also into account that examining a possible configuration in terms of several objectives is also non-trivial. Note that this would still be a polynomial-time algorithm, but in order to quickly react to tenants’ deployment requests, the selection and placement algorithm has to be fast even for large DCs.
Algorithm 1 Determining candidate PMs and VMs

1: procedure CANDIDATES(c)
2: \( V^\ast \leftarrow \emptyset, P^\ast \leftarrow \emptyset \)
3: for all must or should colocation constraint of \( c \) do
4: \( \quad \text{Let } c' \text{ be the other component of the constraint} \)
5: \( \quad \text{if VM-level colocation constraint then} \)
6: \( \quad \quad \text{Add } v(c') \text{ to } V^\ast \)
7: \( \quad \text{else} \quad \quad \triangleright \text{PM-level colocation constraint} \)
8: \( \quad \quad \text{Add } p(v(c')) \text{ to } P^\ast \)
9: \( \quad \quad \text{Add each VM on } p(v(c')) \text{ to } V^\ast \)
10: \( \quad \text{end if} \)
11: \( \text{end for} \)
12: if \( c \) has a VM-based license \( \ell \) then
13: \( \quad \text{for all } v \in V \text{ do} \)
14: \( \quad \quad \text{if there is a component in } v \text{ with license } \ell \text{ then} \)
15: \( \quad \quad \quad \text{Add } v \text{ to } V^\ast \)
16: \( \quad \text{end if} \)
17: \( \text{end for} \)
18: else if \( c \) has a PM-based license \( \ell \) then
19: \( \quad \text{for all } p \in P \text{ do} \)
20: \( \quad \quad \text{if there is a component in } p \text{ with license } \ell \text{ then} \)
21: \( \quad \quad \quad \text{Add } p \text{ to } P^\ast \)
22: \( \quad \quad \text{Add each VM on } p \text{ to } V^\ast \)
23: \( \quad \text{end if} \)
24: \( \text{end for} \)
25: end if
26: for all hardware affinity constraint of \( c \) do
27: \( \quad \text{for all PM } p \text{ with the given PM feature do} \)
28: \( \quad \quad \text{Add } p \text{ to } P^\ast \)
29: \( \quad \quad \text{Add each VM on } p \text{ to } V^\ast \)
30: \( \quad \text{end for} \)
31: \( \text{end for} \)
32: Let \( P^\ast \) be the set of PMs on which \( c \) would fit
33: Sort \( P^\ast \) in decreasing order of CPU load
34: Add the first \( k_1 \) PMs of \( P^\ast \) and their VMs to \( V^\ast \)
35: Add \( k_2 \) random VMs to \( V^\ast \)
36: Add \( k_3 \) random PMs that are on to \( P^\ast \)
37: Add \( k_4 \) random PMs that are off to \( P^\ast \)
38: return \((V^\ast, P^\ast)\)
39: end procedure

For this reason, we decided to first filter the set of candidate PMs and VMs and take only the promising ones into account. As shown in Algorithm 1, we collect the promising VMs in a set \( V^\ast \) and the promising PMs in a set \( P^\ast \). We start by placing a new VM with an appropriate OS in \( V^\ast \) (line 3). If there is a colocation constraint of type must or should between \( c \) and another component \( c' \), then either the VM or the PM hosting \( c' \) is also added, depending on whether it is a VM-level or PM-level colocation constraint (lines 4-12). Such VMs/PMs are indeed promising candidates, since mapping \( c \) onto them would satisfy the colocation constraint. Similarly, if \( c \) has a VM-based license, then all VMs containing a component with the same license are added to \( V^\ast \) (lines 13-18). whereas if \( c \) has a PM-based license, then all PMs containing a component with the same license are added to \( P^\ast \) (lines 19-26). These are again promising since mapping \( c \) onto them would incur no license fee. For hardware affinity constraints of \( c \), all PMs offering the needed feature are added to \( P^\ast \) (lines 27-32). From all PMs where \( c \) would fit without overload, the ones with the highest load are also added to \( P^\ast \) (lines 33-35), since mapping \( c \) onto them would lead
to good utilization and thus to relatively low energy consumption. In all these steps, if a PM $p$ is added to $P^*$, then the VMs hosted on $p$ are added to $V^*$. Finally, we add some further random VMs and PMs to $V^*$ and $P^*$ to extend the search space (lines 36-38). This is important especially if there are few VMs or few dependencies (colocation constraints, common licenses, affinity constraints). At the end, $V^* \times P^*$ defines the set of candidates to examine.

The other challenge in devising the COMBINED algorithm is rooted in the multi-objective nature of the problem: how to determine the best of the examined candidate configurations. As Fig. 5.3 shows, we differentiate between hard factors which should be 0 and soft factors that should be also minimized but with a lower priority than the hard factors.

The factors listed in Fig. 5.3, except for “VM size,” directly relate to costs or constraint violations that need to be minimized. “VM size” has been included because, according to our preliminary experiments, otherwise the algorithm tends to colocate too many components in the same VM. This is logical since — because of the overhead of VMs — mapping the new component to an existing VM is always more energy-efficient than creating a new VM for it. However, having too large VMs may become a disadvantage in the long run, leading to fragmentation of the available PM capacity and hindering the colocation of future components with existing ones even if this were really necessary (because of colocation constraints or license fees). Therefore, since the algorithm makes online decisions based on current objective values without seeing the future, it was necessary to include VM size as a minimization objective to neutralize the energy bias and develop a more future-proof mapping.

For each examined candidate configuration and each optimization objective, we compute the difference that the given selection and/or placement decision would have on the given metric. Based on these atomic metrics, two compound metrics are computed for each examined candidate configuration: the sum of the hard factors and the weighted sum of the soft factors (cf. Fig. 5.3). For the soft factors, weighting is reasonable because power consumption values, license fees, VM sizes and numbers of violations are of different orders of magnitude, so they should be scaled to the same range to allow a meaningful comparison later on. The weight values should thus be chosen depending on the range of license costs, power consumption values etc. The weights can also be used to express differences in the importance of the individual soft factors. For the hard factors, weighting is not necessary (although possible) because all factors are numbers of violations.

To decide whether candidate configuration $x$ is better than candidate configuration $y$, we use the following relation:
Algorithm 2 The COMBINED algorithm for adding a new component $c$

1: $(V^*, P^*) \leftarrow \text{CANDIDATES}(c)$
2: for all $v \in V^*$ do
3:   for all $p \in P^* \cup \{p(v)\}$ do
4:     Compute atomic objectives for $(v, p)$
5:     Compute compound objectives for $(v, p)$
6:   end for
7: end for
8: $(v, p) \leftarrow \text{best examined configuration according to } \prec$
9: if $v$ is a new VM then
10:   Start new VM on $p$
11: else if $p(v) \neq p$ then
12:   Migrate $v$ from $p(v)$ to $p$
13: end if
14: Deploy $c$ on $v$

$x \prec y \iff \text{hard}(x) < \text{hard}(y) \lor (\text{hard}(x) = \text{hard}(y) \land \text{soft}(x) < \text{soft}(y))$,

where $\text{hard}(\cdot)$ and $\text{soft}(\cdot)$ denote the two compound metrics defined above.

Putting all pieces together, Algorithm 2 shows the body of the COMBINED algorithm. It should be noted how VM selection and VM placement are interleaved in this algorithm, since each examined configuration encodes both a VM selection and a VM placement decision.

5.1.3.2 Separate VM selection and VM placement

For comparison, we also develop two policies for VM selection (without VM placement) and two policies for VM placement (without VM selection). Any VM selection policy can be concatenated with any VM placement policy, leading to four different algorithms for deploying a new component.

DEDICATED selection policy  Our first VM selection policy always creates a new, dedicated VM for the new component. Despite its simplicity, this selection policy is quite powerful because it does not create any unnecessary dependence between components, thus leaving full flexibility to the subsequent placement as well as future re-optimizations by live migration. Accordingly, this approach has been used by some previous works [19, 27]. The obvious drawbacks of this policy are the relatively high overhead stemming from the high number of VMs and the lack of colocation for components that must or should be colocated.

INFORMED selection policy  To remedy the shortcomings of the DEDICATED selection policy, we devise a much more sophisticated policy aiming to make a well-informed decision on whether to colocate the new component with existing components or to deploy it in a new VM.

As shown in Algorithm 3, the INFORMED VM selection policy closely resembles the COMBINED algorithm. The differences stem directly from the fact that the INFORMED policy does not account for the placement: hence, it investigates only the possible VMs, not pairs of VMs and PMs. Note also that the INFORMED policy examines all the $|V| + 1$ possible VMs, whereas the COMBINED algorithm had to sample from its much larger search space to remain fast.

The biggest difference is in the way the objectives are computed. From the metrics shown in Fig. 5.3, the “Nr. of PM overloads,” “Nr. of violated must / should hardware affinities,” and “Power consumption” objectives are not applicable at the VM selection stage and are thus ignored in the INFORMED policy. As regards license costs, only VM-based licenses can be considered. VM-level colocation constraints can be fully evaluated, but concerning PM-level colocation constraints, we can only be sure about a violation in case of a must not or should not constraint (if the involved components are mapped to the same VM); for a
Algorithm 3 The INFORMED policy for selecting a VM for the new component $c$

1: Let $v^*$ be a new VM with OS compatible with $c$
2: $V^* \leftarrow V \cup \{v^*\}$
3: for all $v \in V^*$ do
4: Compute atomic objectives for selecting $v$ for $c$
5: Compute compound objectives for selecting $v$ for $c$
6: end for
7: $v \leftarrow$ best examined VM according to $\prec$
8: if $v = v^*$ then
9: Start new VM
10: end if
11: Return $v$

PM-level must or should constraint, a violation cannot be determined at the VM selection stage. The “VM size” metric can be of course fully evaluated.

Because of the — soft — aim of minimizing VM size, components will be only colocated if this is necessary or advantageous for satisfying colocation constraints or for minimizing license fees.

BLACK-BOX placement policy The placement policy receives as input the VM returned by the preceding VM selection policy, which may be a new or an existing VM. The placement policy determines a PM for this VM. In case of an existing VM, this means that the placement policy may decide to migrate the selected VM. This is in line with the COMBINED algorithm, which can also migrate the VM selected for the new component.

The BLACK-BOX placement policy does not consider the components within the VM to place, only its size. This is the same approach as taken by most previous works in the area of VM placement. As suggested by several researchers (e.g., Beloglazov and Buyya [7]), we use the best-fit heuristic to choose the PM that has enough capacity to host the VM but with the minimum remaining free capacity. The VM is then placed on this PM.

Recall that the capacity of a PM is a multi-dimensional vector. For comparing the free capacity of two PMs, we first convert them to single numbers. For this purpose, we take the minimum of the coordinates of the vector. In our previous work we also compared some other metrics for this purpose and found that the minimum metric gives good results [39].

Since this placement policy only considers the size of the VM, we can expect that it will lead to a good placement in terms of energy consumption and number of overloads, but will perform poorly in terms of license costs and conformance with colocation and hardware affinity constraints.

WHITE-BOX placement policy To address the shortcomings of the BLACK-BOX placement policy, we devise a more sophisticated placement policy that also considers the relations of the components within the VM to be placed. Similarly to the INFORMED selection policy, the idea is again to mimic the COMBINED algorithm as much as possible, now at the level of VM placement.

As shown in Algorithm 4, this involves examining all PMs as possible hosts for the VM and choosing the best one in terms of the investigated objectives. From the objectives of Fig. 5.3, now all atomic metrics are relevant except for “VM size.” In terms of license costs, only PM-based licenses are relevant at this stage; similarly, from the colocation constraints, only PM-level constraints are relevant. The other metrics are fully evaluated.

It should be noted that the INFORMED selection policy and the WHITE-BOX placement policy together base their decisions on the same set of information as the COMBINED algorithm and also the way they examine and compare possible candidates is analogous. However, there are two main differences. First, the COMBINED algorithm examines VM-PM pairs, i.e., it considers selection and placement together, whereas
Algorithm 4 The WHITE-BOX policy for placing a VM $v$

1: for all $p \in P$ do  
2: \hspace{1em} Compute atomic objectives for placing $v$ on $p$  
3: \hspace{1em} Compute compound objectives for placing $v$ on $p$  
4: \hspace{1em} end for  
5: $p \leftarrow$ best examined PM according to $\prec$  
6: if $v$ is a new VM then  
7: \hspace{1em} Start new VM on $p$  
8: else if $p \neq p(v)$ then  
9: \hspace{1em} Migrate $v$ from $p(v)$ to $p$  
10: \hspace{1em} end if

in the catenation of INFORMED and WHITE-BOX, first only VMs are considered until one VM is selected, and then only PMs are considered for the already selected VM. This can be seen as an advantage of the COMBINED algorithm. Second, both the INFORMED selection policy and the WHITE-BOX placement policy consider all their possible choices (all VMs respectively all PMs), whereas the COMBINED algorithm only examines a subset of the possible candidate configurations, so that it remains fast. The more thorough search can be seen as an advantage of the INFORMED and WHITE-BOX policies.

5.1.4 Evaluation

Our aim is to compare the different approaches to VM selection and VM placement:

- Decoupled VM selection and VM placement, as in most existing approaches (DEDICATED+BLACK-BOX)
- Partial integration:
  - VM selection also considers VM placement but not vice versa (INFORMED+BLACK-BOX)
  - VM placement also considers VM selection but not vice versa (DEDICATED+WHITE-BOX)
- Semi-integrated: VM selection considers VM placement and vice versa (INFORMED+WHITE-BOX)
- Fully integrated VM selection and VM placement (COMBINED)

5.1.4.1 Setup

Algorithms for VM placement and VM selection are usually evaluated either with a real cloud or by means of simulation. Using a real cloud is of course more realistic but it comes with several limitations. In particular, it is difficult to experiment with many different parameter settings or to scale the size of the experiment if a real cloud is used. Simulations are much more flexible and hence more popular for research on cloud resource management [69] [65] [54] [48]. Since we would like to compare several different algorithms under many different settings, a simulation-based approach is more appropriate. To still obtain practically relevant results, we used real-world test data, leading to a good compromise between a real cloud and pure simulation.

We have implemented all algorithms presented in Section 5.1.3 in a C++ program. To foster reproducibility, this program is freely available from https://sourceforge.net/p/vm-alloc/crosslayer.

In addition to the selection and placement algorithms discussed so far, the program also features a re-optimization algorithm which is invoked regularly and uses VM live migrations to adapt the placement to workload changes. The re-optimization algorithm works as follows: it takes a random VM and uses the WHITE-BOX placement policy to optimize its placement. This optimization step is repeated $k_r$ times, where $k_r$ is a given constant.
Table 5.2: Results for the base setup (component sizes only)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Energy [kWh]</th>
<th>Nr. of overloads</th>
<th>Nr. of migrations</th>
<th>Execution time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMBINED</td>
<td>8,319.35</td>
<td>0</td>
<td>2,390.6</td>
<td>4.0</td>
</tr>
<tr>
<td>DEDICATED+BLACK-BOX</td>
<td>8,570.03</td>
<td>0</td>
<td>1,780.5</td>
<td>0.1</td>
</tr>
<tr>
<td>DEDICATED+WHITE-BOX</td>
<td>8,538.66</td>
<td>0</td>
<td>1,520.2</td>
<td>0.5</td>
</tr>
<tr>
<td>INFORMED+BLACK-BOX</td>
<td>8,567.11</td>
<td>0</td>
<td>1,792.3</td>
<td>0.3</td>
</tr>
<tr>
<td>INFORMED+WHITE-BOX</td>
<td>8,539.47</td>
<td>0</td>
<td>1,496.2</td>
<td>0.9</td>
</tr>
</tbody>
</table>

For component sizes, we used a real workload trace from the Grid Workloads Archive, namely the Auver-Grid trace, available from [http://gwa.ewi.tudelft.nl/datasets/gwa-t-4-auvergrid](http://gwa.ewi.tudelft.nl/datasets/gwa-t-4-auvergrid). From the trace, we used the first 10,000 tasks that had valid CPU and memory usage data. The simulated time (i.e., the time between the start of the first task and the end of the last one) is roughly one month, thus giving sufficient exposure to practical workload patterns.

As PMs, we simulated HP ProLiant DL380 G7 servers with Intel Xeon E5640 quad-core CPU and 16 GB RAM. Their power consumption varies from 280W (zero load) to 540W (full load) [28]. Throughout the experiments, we focus on two resource types: CPU and memory, i.e., \( d = 2 \). Concerning virtualization overhead, previous work reported 5-15\% for the CPU [73] and 107-566 MB for memory [12]. In our experiments, we use 10\% CPU overhead and 300 MB memory overhead. The VM placement is re-optimized every 5 minutes. Similarly, constraint violations are also checked every 5 minutes.

Each reported result is the average of 10 runs.

5.1.4.2 Component sizes only

In our first experiment, components are only characterized by their sizes, i.e., there are no license fees, colocation constraints, nor hardware affinities, and each component has the same OS. This is similar to the evaluation setup of most previous works.

The results – according to the relevant metrics – are shown in Table 5.2. As can be seen, all algorithms result in 0 overloads. In terms of energy consumption, the COMBINED algorithm has a clear advantage over the others; the results of the others are very close to each other. In particular, the used selection policy has practically no effect. This is indeed true because in this case the INFORMED policy has no reason to colocate multiple components in the same VM, hence it also starts a dedicated VM for each component. The WHITE-BOX placement policy performs slightly better than the BLACK-BOX policy. Since the components are characterized only by their sizes, there is not much difference between the two placement policies. The difference is only that BLACK-BOX uses the best-fit heuristic whereas WHITE-BOX chooses the PM based on its real power consumption.

The advantage of the COMBINED algorithm over the second best in terms of energy consumption is 219.31 kWh, or roughly 2.6\%. The average electricity price in the Euro area for industrial customers amounted to 0.125 Euro per kWh in 2015 [4] Thus, the savings translate to 27.1 Euro. Scaling it to a data center with 10 thousand PMs, considering a 12-month period, and assuming a PUE (power usage effectiveness) of 1.7, which is a typical value [5], the total savings would amount to over 240,000 Euro per year.

In terms of the number of migrations (fourth column of Table 5.2), there is a clear difference between the algorithms. This could be important because too many migrations could lead to performance degradation or even make the system unstable [56, 16]. However, relative to the length of the simulation, the number of migrations is actually quite low for all algorithms; even for the COMBINED algorithm which leads to the highest number of migrations, the average number of migrations per PM per day is only 3.32, which should not cause any problems.
Table 5.3: Number of constraint violations plus number of overloads for different colocation constraints

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>PM-level</th>
<th>VM-level</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>must</td>
<td>should</td>
<td>should not</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMBINED</td>
<td>1,956.9</td>
<td>140.1</td>
<td>0</td>
</tr>
<tr>
<td>DEDICATED+BLACK-BOX</td>
<td>12,541.8</td>
<td>12,321.9</td>
<td>1,572.9</td>
</tr>
<tr>
<td>INFORMED+BLACK-BOX</td>
<td>14,878.7</td>
<td>13,587.3</td>
<td>1,615.5</td>
</tr>
<tr>
<td>INFORMED+WHITE-BOX</td>
<td>13,416.7</td>
<td>11,141.3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>13,752.2</td>
<td>10,585.6</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17,473.9</td>
<td>13,473.9</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>8,969.7</td>
<td>7,587.0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>6,116.5</td>
<td>6,258.3</td>
<td>0</td>
</tr>
</tbody>
</table>

Similarly, the COMBINED algorithm takes considerably more time (see last column of Table 5.2) than the other methods, but with an average execution time of 4.0 milliseconds, it can be still considered fast enough.

5.1.4.3 Colocation constraints

The next set of experiments evaluates the impact of colocation constraints. In each experiment, 100 colocation constraints were generated for randomly selected components. Table 5.3 shows the results in a condensed form. Each column corresponds to one experiment. For example, in the experiment of the second column, all colocation constraints were PM-level and of type must; in the 8th column, all colocation constraints were VM-level and of type should not. While in most columns, all colocation constraints were on the same level and of the same type, the last column is different: it is a mix of the 8 combinations of colocation level and type, where each combination is present with approximately the same number of constraints. For each experiment, we report the sum of the number of colocation constraint violations and the number of overloads.

For PM-level must and should colocation constraints (second and third column of the table), the COMBINED algorithm is clearly superior to all others, and all other algorithms achieve similarly poor results. Looking more precisely into the operation of the algorithms, the following can be understood about the reasons:

- The INFORMED selection policy has no incentive to colocate multiple components in the same VM since in these experiments the colocation constraints are all on the PM level. As a result, it creates a dedicated VM for each component. This is why there is no significant difference between the results of the two VM selection policies.

- Since both VM selection policies create small VMs, this leads to low fragmentation. Therefore, when a PM-level colocation constraint motivates the WHITE-BOX placement algorithm to place the new VM on the PM where one of the already placed components resides, it will often not succeed because the given PM does not have sufficient free capacity. This is why there is no significant difference between the results of the two VM placement policies.

- The COMBINED algorithm on the other hand, when confronted with this situation, will put the new component into the same VM as its peer and then migrate the VM containing both components to a PM with sufficient free capacity. Note that the other approaches also have this option, but do not choose it because of the separate evaluation of the selection and placement possibilities.

Concerning the PM-level should not and must not constraints, the results are more easily understood. The COMBINED algorithm as well as the WHITE-BOX placement policy are able to avoid constraint violations by not placing the new component (respectively the VM where it has been put) onto the same PM as some other component(s). The BLACK-BOX placement policy, which does not consider colocation constraints, necessarily leads to some violations. It is interesting to note that the number of violations is now much lower than in the case of must and should constraints. This is not surprising though: placing the new VM
on a random PM has a high chance to meet a should not or must not constraint if the number of “bad” PMs is low, whereas meeting a must or should constraint has much lower probability.

For VM-level must and should colocation constraints, the DEDICATED VM selection policy obviously leads to poor results since it never selects the same VM for components that should be colocated. In fact, these results are even significantly worse than in the case of the similar PM-level constraints, since in this case definitely all constraints will be violated, whereas in the case of PM-level constraints, PM-level colocation was still possible. The INFORMED selection policy, on the other hand, puts all components that must or should be in the same VM indeed into the same VM. Together with the WHITE-BOX placement policy, this leads to very good results, similar to those of the COMBINED algorithm. For VM-level must colocation constraints, it even improves on the results of the COMBINED algorithm.

For VM-level should not and must not colocation constraints, as can be seen, all tested algorithms achieve optimal results. Not colocating some components in the same VM is very easy, for example by using dedicated VMs for each component.

Over all experiments with colocation constraints, the COMBINED algorithm gives excellent results: with the exception of the VM-level must constraints, where it ranks only second after the INFORMED+WHITE-BOX combination, it always gives the best results, in several cases dramatically better results than any other algorithm. This is also mirrored in the last column of Table 5.3 showing the combined effect of different colocation constraints. Here, too, the COMBINED algorithm is the clear winner, leading to more than an order of magnitude better results than all other algorithms. From the results of the remaining algorithms it is also apparent that the INFORMED VM selection policy has a clear advantage over the DEDICATED policy. This is not surprising, given the inability of the DEDICATED policy to appropriately handle VM-level must or should colocation constraints.

Further details and empirical results can be found in the attached TPDS 2018 paper.

5.1.5 Summary of findings

Based on the measurement results of Section 5.1.4, we can now try to answer the original questions regarding the benefits of (i) doing VM selection and VM placement together and (ii) including information about VM placement in VM selection and about VM selection in VM placement. For this purpose, we first summarize the empirical results:

- For PM-based licenses, the COMBINED algorithm results in up to 44% lower license fees than the second best algorithm.

- For VM-based licenses, the COMBINED algorithm and the INFORMED VM selection policy lead to significantly lower license fees than the DEDICATED policy. However, in the case of the INFORMED policy, this comes at the cost of PM overloads.

- For PM-level must and should colocation constraints, COMBINED leads to respectively 84% and 99% improvement over the second best algorithm.

- For PM-level should not and must not colocation constraints, the COMBINED algorithm and the WHITE-BOX VM placement policy lead to clearly better results than the BLACK-BOX placement policy.

- For VM-level must and should colocation constraints, the COMBINED algorithm and the INFORMED+WHITE-BOX algorithm lead to considerably better results than the other algorithms.

- In the presence of hardware affinity constraints, the COMBINED algorithm and the WHITE-BOX VM placement policy result in up to 97% better results than the BLACK-BOX placement policy.
• When all types of constraints were applied at once, the BLACK-BOX policy performed poorly; from the other algorithms, COMBINED was clearly the best, INFORMED+WHITE-BOX was second, and DEDICATED+WHITE-BOX was the third according to all considered dimensions.

• The COMBINED algorithm leads to 2-3% lower energy consumption than the four other tested methods. (Although not shown explicitly, this holds consistently for all cases without overloads.)

Altogether, we can conclude that the COMBINED algorithm is in all cases among the best-performing algorithms. Sometimes it clearly outperforms all other methods (e.g., for PM-based licenses), in other cases it is just one of the best performers. However, in none of the test cases was it clearly inferior to another algorithm. So the answer to the first question is clear: combining VM selection and VM placement in a single optimization algorithm leads to significant benefits compared to the isolated treatment of the two problems as in most existing works (DEDICATED+BLACK-BOX).

Regarding the second question, we can state that the WHITE-BOX VM placement policy clearly outperforms the BLACK-BOX policy. Hence, it is advantageous to include component-level information in VM placement decisions. The relationship between the DEDICATED and INFORMED VM selection policies is less clear because in many cases, the difference between their results was marginal. However, there were some test cases where INFORMED led to clearly better results: for VM-level must and should colocation constraints and, even more importantly, when all types of constraints were applied at once. Thus we conclude that, for these types of scenarios, it is also advantageous to include in VM selection decisions foresight into VM placement.

In terms of the current state of the art, it should be underlined that existing approaches almost exclusively focus either on VM selection or VM placement, ignoring the other problem. Hence, the DEDICATED+BLACK-BOX algorithm embodies the current state of the art. Compared to this, our results show that the tighter the two problems are integrated, the better results can be achieved.

The comparison between the fully integrated approach (COMBINED) and the semi-integrated approach (INFORMED+WHITE-BOX) is tricky because none of them dominates the other in all considered metrics. COMBINED was better in handling licenses and PM-level colocation constraints, the two approaches yielded similar results for handling VM-level colocation constraints and HW affinity constraints, while INFORMED+WHITE-BOX led to fewer migrations and had lower execution time. The number of migrations and the execution time of COMBINED are also in an acceptable range, and so, since it leads to lower license costs and fewer constraint violations, we consider it preferable. Also in the experiment which contained all the investigated aspects, the COMBINED approach led to better results according to all measured metrics.

A further question is whether there is any “hidden” overhead of the proposed approach, since it leads to more co-locations and as VM isolation is not perfect, this could lead to performance degradation. Fortunately, our model supports anti-colocation constraints with which co-location of components that would interfere with each other can be prohibited. Therefore, the proposed approach will only co-locate components that do not interfere.

5.2 Optimized secure deployment of software components in hybrid clouds

For deploying software applications, companies can use on-premise computing facilities in the form of private clouds or they can employ public cloud offerings, such as Infrastructure-as-a-Service (IaaS) \[57\]. Both deployment options have advantages and disadvantages. Public clouds offer unparalleled flexibility in terms of their ability to quickly launch even large numbers of virtual machines, providing practically unlimited capacity \[11\]. This is a big advantage, especially to cope with temporarily increased demand, which would be a problem with a fixed set of on-premise resources. On the other hand, using public cloud resources may entail data protection risks, including attacks by employees of the cloud provider or side-channel attacks by other tenants with co-located virtual machines \[3\]. In this regard, on-premise resources are preferable, since the owner organization has full control over them \[67, 22\].
Regarding financial costs, there is no general answer as to whether on-premise hosting in the private cloud or using public cloud services is cheaper. Previous analyses showed that this depends on many factors \[61, 32\]. We assume that a specific amount of on-premise computing capacity is already purchased and operated anyway (e.g., because of the data protection considerations mentioned above). Therefore, the costs of this on-premise computing capacity are already covered, so that adding new load is practically for free – as long as allowed by the capacity. On the other hand, using public cloud resources incurs direct financial costs both for computing power and for data transfers to and from the public cloud. Hence from the financial perspective, it is beneficial to use the in-house resources as much as possible and only use the public cloud for excess load that cannot be handled by the limited on-premise capacity – an approach also known as cloud bursting [23].

These considerations lead to an optimization problem: given a set of applications to be deployed, how should those applications be partitioned between the private and the public cloud, so that (i) all data-protection-relevant application components are placed into the private cloud, (ii) the capacity of the private cloud is not exceeded, and (iii) the cost of cloud usage is minimized? It should be noted that (i) and (ii) are hard constraints that must be met by all means, whereas in the case of (iii) the goal is to have costs as low as possible but optimality is not a hard requirement.

Optimizing the deployment of the applications is not a one-off activity. Rather, the deployment should be re-optimized regularly during operation, for instance when a new application is added or an existing application is removed, when the load on an application changes, when cloud prices change, etc. Therefore, the optimization algorithm must be fast enough for online operation.

We will show that the defined optimization problem is NP-hard. Existing algorithms for this or similar problems are either exact algorithms (e.g., based on integer linear programming) or heuristics [36]. Exact algorithms are typically too slow to be used online with practical problem sizes. Existing heuristics are typically either not powerful enough to achieve near-optimal costs, or cannot guarantee that the found solution will always fulfill the required properties. Thus it remains a challenge to devise an appropriate algorithm for this problem.

Here, we describe a carefully designed algorithm for this problem. SODA (Secure Optimized Deployment Algorithm) is a fast heuristic based on the Kernighan-Lin graph partitioning algorithm that iteratively improves an existing partition by a sequence of local changes, while also being able to escape local optima [31]. The algorithm is a heuristic and thus it is not guaranteed that its results would be optimal in terms of cloud usage costs. However, as we show, the result of the algorithm is guaranteed to satisfy the security and capacity requirements, whenever they can be satisfied. Moreover, our experimental findings show that the costs of the deployment found by SODA are very near to the optimum.

5.2.1 Problem

In this section, we first informally introduce the problem and then provide a formal problem model and formulation, as well as a complexity analysis of the problem.

5.2.1.1 Informal problem description

To illustrate the problem, we consider a company that wants to run a set of applications in a hybrid cloud, consisting of a private and a public cloud (see Fig. 5.4). Using the private cloud is considered to be secure, as the company has full control about who has access to the resources in the private cloud. However, the capacity of the private cloud is limited. In contrast, the public cloud is considered to have unlimited capacity but it is less secure, since the provider of the public cloud or other tenants might get access to software components deployed in the public cloud [3]. Moreover, using the public cloud is associated with costs, both for the usage of compute resources and for data transfers between the private and the public cloud.
Each application consists of a set of components. Each component requires a given number of processor cores. Some components are marked as sensitive: these are the components that store or process sensitive data and hence must remain in the private cloud to avoid data breach. Application components exchange data across links. A link may be within a cloud or inter-cloud, depending on whether the two components that the link connects are deployed in the same cloud. Components are deployed in containers or virtual machines, which makes it possible to easily move the components between the private and the public cloud if necessary.

Sometimes, a new application is deployed, or an already deployed application is removed. Deployed applications may undergo changes, affecting the CPU requirements of components, the amount of data transfer between components, or whether a component is sensitive or not. Also the unit price of using public cloud resources may change. After such events, the deployment of all applications may be re-optimized. The goal is to minimize the financial costs stemming from using the public cloud, subject to two constraints: (i) the total number of CPU cores required by the components deployed in the private cloud must not exceed the capacity of the private cloud, and (ii) all sensitive components must be deployed in the private cloud.

5.2.1.2 Formal problem model

The set of applications is denoted by \( \mathcal{A} \) (see also Table 5.4). Each application \( A \in \mathcal{A} \) is represented by an undirected graph \( A = (V_A, E_A) \), where \( V_A \) is the set of components of application \( A \) and \( E_A \) is the set of links among the components. The set of all components in all applications is \( V = \bigcup \{ V_A : A \in \mathcal{A} \} \). Similarly, the set of all links among components in all applications is \( E = \bigcup \{ E_A : A \in \mathcal{A} \} \).

For a component \( v \in V \), \( p(v) \in \mathbb{N} \) is the number of processor cores required by the component. The predicate \( s(v) \) is true if and only if \( v \) is sensitive. For a link \( e \in E \), \( h(e) \in \mathbb{R}^{+} \) denotes the amount of data exchanged along the link.

\( P \in \mathbb{N} \) denotes the number of processor cores available in the private cloud. The cost of renting a processor core in the public cloud is denoted by \( c_1 \). The cost of transferring a unit amount of data between the private and the public cloud is denoted by \( c_2 \).

A deployment is a function \( d : V \rightarrow \{ \text{priv}, \text{publ} \} \) that maps each component to either the private or the public cloud. We use \( g(d) \) to denote the total number of CPU cores used in the private cloud by deployment \( d \):

\[
g(d) = \sum_{v \in V; d(v) = \text{priv}} p(v).
\]
Table 5.4: Notation overview

<table>
<thead>
<tr>
<th>Notation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Set of applications</td>
</tr>
<tr>
<td>$V_A$</td>
<td>Set of components of application $A$</td>
</tr>
<tr>
<td>$E_A$</td>
<td>Set of links among components of application $A$</td>
</tr>
<tr>
<td>$V$</td>
<td>Set of all components</td>
</tr>
<tr>
<td>$E$</td>
<td>Set of all links among components</td>
</tr>
<tr>
<td>$p(v)$</td>
<td>Number of CPU cores required by component $v$</td>
</tr>
<tr>
<td>$s(v)$</td>
<td>True if and only if component $v$ is sensitive</td>
</tr>
<tr>
<td>$h(e)$</td>
<td>Amount of data exchange through link $e$</td>
</tr>
<tr>
<td>$P$</td>
<td>Number of CPU cores available in the private cloud</td>
</tr>
<tr>
<td>$c_1$</td>
<td>Unit cost of renting CPU cores in the public cloud</td>
</tr>
<tr>
<td>$c_2$</td>
<td>Unit cost of inter-cloud data transfer</td>
</tr>
<tr>
<td>$d$</td>
<td>Deployment function</td>
</tr>
<tr>
<td>priv</td>
<td>Label for components in the private cloud</td>
</tr>
<tr>
<td>publ</td>
<td>Label for components in the public cloud</td>
</tr>
<tr>
<td>$g(d)$</td>
<td>Nr. of CPU cores deployment $d$ uses in the private cloud</td>
</tr>
<tr>
<td>$E(d)$</td>
<td>Set of inter-cloud links, given deployment $d$</td>
</tr>
</tbody>
</table>

A valid deployment must respect the following constraints:

$$g(d) \leq P,$$

$$\forall v \in V : s(v) \Rightarrow (d(v) = \text{priv}).$$

Constraint (5.1) ensures that the total number of CPU cores required by the components allocated to the private cloud does not exceed the number of CPU cores available in the private cloud. Constraint (5.2) ensures that all sensitive components are deployed in the private cloud.

The two constraints may lead to a contradiction if the private cloud does not have enough CPU capacity to host all sensitive components, but otherwise, the constraints are satisfiable. This is captured by the following

**Proposition 1.** A deployment exists that satisfies both constraints (5.1) and (5.2) if and only if

$$\sum_{v \in V, s(v)} p(v) \leq P.$$ 

**Proof.** If a deployment $d$ satisfies both constraints (5.1) and (5.2), then

$$\sum_{v \in V, s(v)} p(v) \leq \sum_{v \in V, d(v)=\text{priv}} p(v) \leq P$$

(the first inequality follows from (5.2) and $p(v) \geq 0$, while the second inequality follows from (5.1)), which proves (5.3).

To see the reverse, assume that (5.3) holds, and let

$$d(v) = \begin{cases} 
\text{priv}, & \text{if } s(v); \\
\text{publ}, & \text{otherwise.} 
\end{cases}$$

(5.4a)

Here, (5.4a) ensures that $d$ satisfies (5.2). On the other hand,

$$\sum_{v \in V, d(v)=\text{priv}} p(v) = \sum_{v \in V, s(v)} p(v) \leq P$$

(the equation follows from the definition of $d$, the inequality follows from (5.3)), which proves that $d$ also satisfies (5.1).
In the following, we will assume that (5.3) holds so that the problem of finding a valid deployment is solvable. Our aim is to find a solution that minimizes the financial cost. For a deployment $d$, the set of inter-cloud links is defined as $E(d) = \{ uv \in E : d(u) \neq d(v) \}$. Then, the cost of deployment $d$ is defined as follows:

$$\text{cost}(d) = \sum_{v \in V, d(v) = \text{publ}} c_1 \cdot p(v) + \sum_{e \in E(d)} c_2 \cdot h(e),$$

(5.5)

where the first term is the total cost of leased compute resources in the public cloud and the second term is the total cost of data transfers between the private and public clouds.

The problem of minimizing (5.5) while satisfying (5.1) and (5.2) will be called the minimum-cost deployment problem.

5.2.1.3 Complexity Analysis

The NP-hardness of this or similar problems was claimed multiple times in the literature [67, 68]. However, we are not aware of a proof. In the following, we prove the stronger claim that the problem is strongly NP-hard, meaning that it is NP-hard even in the special case when all numbers in the problem are polynomially bounded with respect to the problem size [46].

**Theorem 2.** The minimum-cost deployment problem is strongly NP-hard.

The proof is omitted for brevity.

As a consequence of strong NP-hardness, we cannot expect a polynomial-time, nor even a pseudo-polynomial-time exact algorithm, nor a fully polynomial-time approximation scheme for this problem, under the standard assumptions of complexity theory [46].

5.2.1.4 Problem formulation with integer linear programming

We now show an integer linear programming (ILP) formulation of the problem. This serves two purposes: (i) it is a further formalization of the problem and (ii) it can be solved directly by an appropriate ILP solver, yielding an exact algorithm for the problem which serves as a baseline for experimental evaluation.

We define two sets of binary variables $\{x_v : v \in V\}$ and $\{y_e : e \in E\}$ with the following interpretation:

$$x_v = \begin{cases} 0 & \text{if } d(v) = \text{priv} \\ 1 & \text{if } d(v) = \text{publ} \end{cases}$$

$$y_e = \begin{cases} 0 & \text{if } e \not\in E(d) \\ 1 & \text{if } e \in E(d) \end{cases}$$

The ILP can be formulated as follows:

$$\min c_1 \cdot \sum_{v \in V} p(v) \cdot x_v + c_2 \cdot \sum_{e \in E} h(e) \cdot y_e$$

(5.6)

s.t.

$$\sum_{v \in V} p(v) \cdot (1 - x_v) \leq P$$

(5.7)

$$x_v = 0$$

if $s(v)$

(5.8)

$$x_v - x_w \leq y_{vw}$$

$\forall vw \in E$  \hspace{1cm} (5.9)

$$x_w - x_v \leq y_{vw}$$

$\forall vw \in E$  \hspace{1cm} (5.10)

The objective function (5.6) corresponds to the cost function (5.5) defined earlier, while constraints (5.7) and (5.8) correspond to the constraints (5.1) and (5.2) defined earlier. The constraints (5.9) and (5.10)
ensure that the values of the $x$ and $y$ variables are consistent with each other; in particular, they ensure that if $x_v \neq v_w$, then $y_{vw} = 1$. If $x_v = x_w$, then the value of $y_{vw}$ is not constrained; however, since $y_{vw}$ has a positive weight in the objective function, in any optimal solution of the ILP $y_{vw} = 0$ will hold.

5.2.2 Algorithm

In this section we describe SODA (Secure Optimized Deployment Algorithm), the proposed heuristic algorithm for the minimum-cost deployment problem.

5.2.2.1 Overview

**Algorithm 5** Adding a new application

```
1: procedure ADD(A)
2:     for v ∈ V_A do
3:         if s(v) then
4:             d(v) ← priv
5:         else
6:             d(v) ← publ
7:         end if
8:     end for
9:     RE-OPTIMIZE(d)
10: end procedure
```

The deployment must be adapted in three situations: (i) when a new application is added, (ii) when an application is removed, (iii) when something changes in the deployed applications or in their environment (e.g., in case of changes of the applications’ workload). When a new application is added, we deploy each component $v$ of the new application with the simple rules given in (5.4a)-(5.4b), and then we re-optimize the deployment (see Algorithm 5). When an application is removed, we simply remove all its components from the deployment, and then perform a re-optimization (see Algorithm 6). When there is a change in the deployed applications (e.g., in the CPU requirements of some components, the amount of data transfer between components, the sensitivity of components) or in the environment (e.g., in the unit price of using the public cloud or the capacity of the private cloud), we first ensure that the security requirement continues to hold, and then perform re-optimization (see Algorithm 7).

Note that all applications compete for the same limited resources in the private cloud, which introduces dependencies among them. When an application is added, the deployment may become invalid (it may violate the capacity constraint, see Section 5.2.1.2). In such a situation, some components of other applications may have to be migrated from the private to the public cloud to make the deployment valid again. If an application is removed, the deployment does not become invalid, but it may be beneficial to migrate some components of other applications from the public to the private cloud to reduce costs. If there is a change in one of the deployed applications, this may lead to a situation that is best fixed by changing the deployment of another application. This is why it makes sense to re-optimize the deployment of all applications each time an application is added or removed or some other change happens.

**Algorithm 6** Removing an application

```
1: procedure REMOVE(A)
2:     for v ∈ V_A do
3:         remove v
4:     end for
5:     RE-OPTIMIZE(d)
6: end procedure
```
Algorithm 7 Handling changes

1: procedure CHANGES
2: for v ∈ V do
3: if s(v) and d(v) = publ then
4: d(v) ← priv
5: end if
6: end for
7: end procedure

Re-optimization is performed in the same way in each of the three cases. Re-optimization is based on iterative improvement, i.e., it starts from a – not necessarily valid – deployment and tries to improve it (i.e., making it valid and decreasing its cost) through a series of local changes. In each step, one component is moved either from the private cloud to the public cloud or vice versa. Note that these moves are not real migrations, they are just performed tentatively on a graph model used internally by the algorithm. The algorithm is greedy in the sense that it always makes the move that seems best. If the current deployment violates the capacity constraint, a component is moved from the private to the public cloud; otherwise, the algorithm moves the component leading to the highest decrease in deployment cost. The quantity that forms the basis for decision-making is called the gain of the components and is defined as follows.

Definition 3. Let d be a deployment and v ∈ V a component. Let d’ be the deployment obtained from d by moving v to the other cloud, i.e., for a component w ∈ V,

\[
d’(w) = \begin{cases} 
    d(w) & \text{if } w \neq v, \\
    \text{priv} & \text{if } w = v \text{ and } d(v) = \text{publ}, \\
    \text{publ} & \text{if } w = v \text{ and } d(v) = \text{priv}.
\end{cases}
\]

Then, given deployment d, the gain of moving v is defined as

\[
\text{gain}(d, v) = \begin{cases} 
    -\infty & \text{if } d \text{ is valid, } d’ \text{ is invalid}, \\
    \text{cost}(d) - \text{cost}(d’) & \text{otherwise}.
\end{cases}
\]

The algorithm prefers moves with higher gain values. To escape local optima, the move with highest gain is made even if its gain is negative, i.e., it increases the cost (except if the gain is −∞). To avoid infinite loops, each component may be moved only once. When no further move is possible, the deployment with the lowest cost that was encountered during the algorithm is taken as the resulting new deployment, and the necessary migrations are carried out to actually achieve the desired deployment.

The re-optimization procedure used in SODA is an extended version of the Kernighan-Lin algorithm for balanced graph partitioning [31]. The Kernighan-Lin algorithm and its variants have been successfully applied to several different partitioning problems, see for example [42] and references therein. Its success can be attributed to the fact that, while it is a fast heuristic, it can still escape local optima. Applying the Kernighan-Lin algorithm to our optimization problem required several extensions, since the original algorithm supports only edge costs, whereas our problem also contains costs related to nodes, as well as hard constraints on capacity and security, which are also not supported by the original algorithm.

5.2.2.2 Detailed description of deployment re-optimization

A more formal description of the re-optimization procedure is given in Algorithm 8. The algorithm starts by setting “best_deployment” and “best_cost” to the current deployment respectively its cost (lines 2-3). The list L contains the components that may be moved. In line 4, L is initialized to the set of all non-sensitive components; the sensitive components are not movable since they must remain in the private cloud. In each
Algorithm 8 Deployment re-optimization

1: procedure RE-OPTIMIZE(d)
2:   best_deployment ← d
3:   best_cost ← cost(d)
4:   L ← {v ∈ V : ¬s(v)}
5:   end ← (L = ∅)
6: while ¬end do
7:   best_gain ← −∞
8:   for v ∈ L do
9:     if g(d) ≤ P or d(v) = priv then
10:        g ← GAIN(d,v)
11:        if g > best_gain then
12:           best_comp ← v
13:           best_gain ← g
14:        end if
15:   end if
16: end for
17: if best_gain > −∞ then
18:   forced ← (g(d) > P)
19:   change d(best_comp) to the other value
20:   L.remove(best_comp)
21: if forced or cost(d) < best_cost then
22:   best_deployment ← d
23:   best_cost ← cost(d)
24: end if
25: end if
26: end while
27: end procedure

In each iteration, first the component to be moved is determined (“best_comp”). For that purpose, “best_gain” is initialized to −∞ (line 8), and then all movable components are checked (lines 8-16). Line 10 ensures that moving a component from the public to the private cloud is not considered if the private cloud is already overloaded. Lines 10-14 serve the purpose of determining the component with the highest gain. If an allowed move has been found, then it is performed (line 19) and the corresponding component is removed from L (line 20). If the private cloud was overloaded before the move, then the move is forced to be from the private to the public cloud, as captured by the Boolean variable “forced”. In this case, “best_deployment” and “best_cost” are certainly updated with the changed deployment and its cost, otherwise they are updated only if the changed deployment is better than the best deployment encountered so far in terms of costs (lines 18, 21-24). The loop ends if there are no more movable components (L = ∅) or there are no valid moves, i.e., there are only moves that would invalidate the deployment (“best_gain” = −∞) (line 26). Finally, the best deployment found is chosen (line 28).

A further important detail is how the gain of moving a component is calculated. The gain is computed by Algorithm 9 in line with Definition 3. If the component v is currently in the private cloud, then moving it to the public cloud would increase costs by c1 · p(v) (lines 2-3). Similarly, if it is in the public cloud, then moving it to the private cloud would decrease costs by the same amount (lines 6-7). However, if the move would violate the capacity constraint of the private cloud, then the move is not allowed, resulting in a gain of −∞ (lines 4-5). In lines 9-15, the links incident to the component v are investigated. For a link vw, if v and w are in the same cloud, then the move would result in vw crossing the boundary between the two clouds.
Algorithm 9 Calculation of the gain of moving a component

1: procedure GAIN(d, v)
2: if \(d(v) = \text{priv}\) then
3: \(r \leftarrow -c_1 \cdot p(v)\)
4: \(\text{else if } g(d) \leq P \quad \text{and } g(d) + p(v) > P\) then
5: \(\text{return } -\infty\)
6: \(\text{else}\)
7: \(r \leftarrow c_1 \cdot p(v)\)
8: \(\text{end if}\)
9: for \(vw \in E\) do
10: if \(d(v) = d(w)\) then
11: \(r \leftarrow r - c_2 \cdot h(vw)\)
12: \(\text{else}\)
13: \(r \leftarrow r + c_2 \cdot h(vw)\)
14: \(\text{end if}\)
15: \(\text{end for}\)
16: \(\text{return } r\)
17: \(\text{end procedure}\)

clouds, increasing the cost by \(c_2 \cdot h(vw)\) (lines 10-11). Otherwise, i.e., if the two components are in different clouds, then moving \(v\) would result in \(vw\) not crossing the inter-cloud boundary anymore, thus decreasing the cost by the same amount (lines 12-13).

5.2.3 Analysis

In this section, we analyze the execution time of our SODA algorithm and prove the correctness of the algorithm.

5.2.3.1 Time complexity of SODA

First, we analyze the execution time of the re-optimization algorithm.

**Theorem 4.** The time complexity of Algorithm 8 is \(O(|V| \cdot (|V| + |E|))\).

*Proof.* The time complexity of Algorithm 8 is dominated by the *while* loop (lines 6-27) and the nested *for* loop (lines 8-16). In each pass of the *while* loop, one component is removed from \(L\) (“best_gain” \(> -\infty\), line 20) or the loop terminates (“best_gain” \(= -\infty\), line 26); therefore, the *while* loop is repeated at most \(|V|\) times. In the *for* loop, each component in \(L\) is considered; in the worst case, this means all components in \(V\). For a component \(v\), the gain calculation takes \(O(1 + \delta(v))\) steps, where \(\delta(v)\) is the number of links incident to \(v\). The remaining steps in the body of the *for* loop take \(O(1)\) steps. Hence, the total time spent in the *for* loop is at most \(\sum_{v \in V} O(1 + \delta(v)) = O(|V| + |E|)\). Together with the already established fact that the *while* loop is repeated at most \(|V|\) times, this completes the proof.

**Corollary 5.** The time complexity of Algorithms 5, 6, and 7 is also \(O(|V| \cdot (|V| + |E|))\).

*Proof.* For Algorithms 5, 6, and 7 all steps before the re-optimization together take \(O(|V|)\) steps. Thus the time complexity of these algorithms is dominated by the re-optimization.

5.2.3.2 Correctness of SODA

To prove the correctness of our algorithms, we reason about sequences of algorithm calls.
Table 5.5: Components of CoCoME case study

<table>
<thead>
<tr>
<th>Name</th>
<th>Required CPU cores</th>
<th>Sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>storeManager</td>
<td>2</td>
<td>yes</td>
</tr>
<tr>
<td>bankInterface</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>loyalty</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>reporting</td>
<td>2</td>
<td>no</td>
</tr>
<tr>
<td>inventory</td>
<td>4</td>
<td>no</td>
</tr>
<tr>
<td>pickupShop</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>dataManager</td>
<td>2</td>
<td>no</td>
</tr>
<tr>
<td>productManager</td>
<td>2</td>
<td>no</td>
</tr>
</tbody>
</table>

Definition 6. A sequence of calls is a sequence $\Gamma = (\gamma_1, \gamma_2, \ldots, \gamma_k)$, where each $\gamma_i \in \{\text{add, remove, change}\}$, depending on whether the $i$th call was to Algorithm 5 (addition of a new application), Algorithm 6 (removal of an application), or Algorithm 7 (other change happened). The set of applications and the deployment after $i$ calls are denoted by $A^{(i)}$ and $d^{(i)}$, respectively. In particular, $A^{(0)}$ and $d^{(0)}$ denote the set of applications respectively the deployment before the first call.

Theorem 7. Starting from $A^{(0)} = \emptyset$ and performing an arbitrary sequence of calls $\Gamma = (\gamma_1, \gamma_2, \ldots, \gamma_k)$, if condition (5.3) is satisfied throughout (i.e., the deployment problem is solvable), then each call results in a valid deployment.

The proof is omitted for brevity.

We have established that our algorithms will always return deployments that satisfy the hard constraints on capacity and security, whenever this is possible. Even if the problem becomes unsolvable, the result will be as useful as possible:

Theorem 8. If (5.3) does not hold, then the output of a call to Algorithm 5, Algorithm 6, or Algorithm 7 is the deployment in which $(d(v) = \text{priv}) \iff s(v)$. In other words, (5.2) is satisfied even in this case, and the deviation from (5.1) is smallest possible.

The proof is omitted for brevity.

Concerning costs, the algorithm aims for minimization, but there is no guarantee that the delivered deployment will have minimum costs. Experimental results on the resulting costs are presented in Section 5.2.5.

5.2.4 Case study

To assess the practical applicability of the proposed approach, we implemented it as a Java program. As a comparison baseline, we also implemented the ILP-based approach using the ILP formulation from Section 5.2.1.4 in the same Java program. Moreover, we also implemented a simple heuristic based on the first-fit principle. All three algorithms can be used to solve the same problem instances and their results can be compared. To foster the reproducibility of the results, we made the program publicly available.

To demonstrate the applicability of our approach and illustrate its operation, we applied it to the cloud-based variant of the CoCoME case study [26]. CoCoME models cloud services that support the typical trading operations of a supermarket chain, like the management of stores, inventory management, and product dispatching. As such, CoCoME offers a realistic case study covering computationally intensive application components, data transfers among the components, and data protection concerns.

The characteristics of the components of the CoCoME application are shown in Table 5.5; the characteristics of the links among the components are shown in Table 5.6. The unit costs of compute resources and of

---

*https://sourceforge.net/p/vm-alloc/hybrid-deployment/*
Table 5.6: Links and data transfers in CoCoME case study

<table>
<thead>
<tr>
<th>Link (endpoint1 ↔ endpoint2)</th>
<th>Data transfer [GB/day]</th>
</tr>
</thead>
<tbody>
<tr>
<td>storeManager ↔ bankInterface</td>
<td>0.04</td>
</tr>
<tr>
<td>storeManager ↔ reporting</td>
<td>0.5</td>
</tr>
<tr>
<td>storeManager ↔ inventory</td>
<td>1.5</td>
</tr>
<tr>
<td>loyalty ↔ reporting</td>
<td>0.1</td>
</tr>
<tr>
<td>reporting ↔ inventory</td>
<td>3.0</td>
</tr>
<tr>
<td>pickupShop ↔ inventory</td>
<td>0.2</td>
</tr>
<tr>
<td>inventory ↔ dataManager</td>
<td>2.4</td>
</tr>
<tr>
<td>inventory ↔ productDispatcher</td>
<td>0.8</td>
</tr>
</tbody>
</table>

Table 5.7: Gain values of the movable components during the run of Algorithm 8

<table>
<thead>
<tr>
<th>Component</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>reporting</td>
<td>0.89</td>
<td>1.43</td>
<td>–</td>
</tr>
<tr>
<td>inventory</td>
<td>1.77</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>pickupShop</td>
<td>0.53</td>
<td>0.57</td>
<td>–∞</td>
</tr>
<tr>
<td>dataManager</td>
<td>0.89</td>
<td>1.32</td>
<td>–∞</td>
</tr>
<tr>
<td>productDispatcher</td>
<td>1.03</td>
<td>1.18</td>
<td>–∞</td>
</tr>
</tbody>
</table>

inter-cloud data transfer are determined based on Amazon EC2 pricing[7]. The hourly rental fee of a t2.small instance is USD 0.023, leading to a daily fee of USD 0.552, which is used as $c_1$. The transfer of 1GB of data between Amazon EC2 and the Internet costs USD 0.09, which is used as $c_2$.

Assuming 10 available CPU cores in the private cloud, running Algorithm 5 works as follows. First, the sensitive components (storeManager, bankInterface, loyalty) are put into the private cloud and all other components are tentatively put into the public cloud. Then, Algorithm 8 is executed to optimize the deployment. The gain values computed for the movable (i.e., non-sensitive) components are shown in the second column of Table 5.7. As can be seen, the inventory component has the highest gain, in accordance with the fact that it uses the highest number of CPU cores, hence moving it to the private cloud would lead to the highest decrease in costs. Accordingly, the algorithm selects the inventory component and moves it to the private cloud.

In the next step, the gain values of the movable components are re-computed as shown in the third column of Table 5.7. Note that the inventory component is not movable anymore. It is important to note that the gains of the other components changed: this is because of the costs related to the links between the inventory and the other components. In particular, because of the intensive communication between the inventory and reporting components, now the reporting component has the highest gain: after having moved the inventory to the private cloud, it is also beneficial to move the reporting component to the private cloud to avoid the heavy traffic crossing the inter-cloud boundary. Accordingly, the algorithm selects the reporting component and moves it to the private cloud.

In the next step, the gain values of the movable components are re-computed again as shown in the fourth column of Table 5.7. Note that the inventory and reporting components are not movable anymore. All other components have a gain of $-\infty$ since the private cloud is already full (the components already in the private cloud use up all 10 CPU cores), so moving any of the remaining components to the private cloud would violate the capacity constraint. At this moment, the algorithm terminates with the current deployment, which has the lowest cost among the ones investigated.

The deployment created by our algorithm is shown in Fig. 5.5. As can be observed, all sensitive components are deployed in the private cloud, plus two further components, which together use up all 10 CPU

cores. The costs of the deployment are optimal in this case. The ILP-based algorithm delivers the same deployment.

Now let us assume that a further application is deployed, consisting of a single component that is sensitive and uses one CPU core. In effect, this reduces the number of CPU cores available in the private cloud to the CoCoME application from 10 to 9. When re-optimizing the deployment, the algorithm is confronted with an invalid deployment, since the CoCoME application and the newly deployed application would together need 11 CPU cores in the private cloud. Hence, SODA must first make a forced move from the private to the public cloud. There are two movable components in the private cloud: reporting and inventory. Both have a negative gain, but the gain of reporting is higher (it requires fewer CPU cores, hence it costs less in the public cloud), so it is moved to the public cloud. This way, the deployment becomes valid, and there is even a free CPU core in the private cloud. As a consequence, the pickupShop component, which needs only one CPU core, may be moved to the private cloud, thus it has a positive gain. When the pickupShop component is moved to the private cloud, the deployment reaches a local optimum. The only valid move is to move the inventory to the public cloud. This is a worsening move, but the algorithm carries it out. As it turns out, this worsening move pays off because it frees up 4 CPU cores in the private cloud, so that in the next two steps the productDispatcher and dataManager components can be moved to the private cloud. The resulting deployment is better than the local optimum found along the way, thanks to the heavy traffic between reporting and inventory that now does not have to cross the inter-cloud boundary anymore. The resulting deployment can be seen in Fig. 5.6 which also shows the order in which the components are moved by SODA. This deployment is again optimal since the ILP-based exact algorithm also delivers the same result.

These examples illustrated how the proposed algorithm ensures satisfaction of the requirements, and at the same time uses the remaining degrees of freedom to optimize costs. Moreover, the second example showed how SODA can escape local optima.

5.2.5 Experimental evaluation

In this section we present experimental results about the costs of the solutions delivered by the proposed algorithm as well as its execution time. We compare the performance of SODA to two competing algorithms:

- The ILP-based algorithm, using the ILP formulation from Section 5.2.1.4 as a typical example of an exact algorithm.
A simple heuristic based on the first-fit (FF) principle, as a typical example of a greedy algorithm. It works as follows:

1. It places all sensitive components into the private cloud.
2. For each of the remaining components, if that component still fits into the private cloud, then it is placed into the private cloud, otherwise into the public cloud.

The experiments were performed on a Lenovo ThinkPad X1 laptop with Intel Core i5-4210U CPU @ 1.70GHz and 8GB RAM. The ILP-based algorithm uses the Gurobi Optimizer, version 7.0.2, as an external solver to solve the ILP problem. The ILP solver was executed in single-threaded mode with a timeout of 60 seconds.

In a first experiment, we simulated the following scenario:

1. 10 applications are randomly generated with the following parameters:
   - $|V_A| = 30$
   - $p(v)$ is uniformly chosen from $\{1, 2, 3, 4\}$ for each $v \in V_A$
   - $s(v)$ is true with probability 0.1 for each $v \in V_A$
   - $(V_A, E_A)$ is a complete graph
   - $h(e)$ is uniformly chosen from $[0.0, 3.0]$ for each $e \in E_A$

2. Starting with $A^{(0)} = \emptyset$, the applications are added one by one in the first 10 steps. Afterwards, 10 change steps are carried out, and finally the applications are removed one by one, leading to $A^{(30)} = \emptyset$. Each change step performs one of the following actions (each with equal probability):
   - For each application, pick 3 random components and either increase or decrease their number of CPU cores by 1.
   - For each application, pick a random component and change its being sensitive.
   - For each application, pick 10 random links and change their traffic intensity by multiplying with 2 or 0.5.
   - Change $c_1$, the unit cost of compute resources, by either increasing or decreasing it by 10%.

Figure 5.6: Modified deployment of the CoCoME application after deploying another application. The numbers show the order in which the components are moved by the algorithm.
Figure 5.7: Results of first adding, then changing, and finally removing 10 applications

- As before, $c_1 = 0.552$ and $c_2 = 0.09$
- $P = 150$

The results of this experiment are shown in Fig. 5.7. Fig. 5.7(a) shows the costs achieved by the three algorithms after each algorithm call. As expected, the costs monotonously increase in the first 10 steps and decrease in steps 20-30. In steps 1-2 and 29-30, it is possible to deploy all components in the private cloud, leading to 0 costs; this optimal deployment is found by all algorithms. In the other steps, some components must be deployed in the public cloud, leading to non-zero costs. Consistently across all steps 3-28, the results of SODA are only slightly higher than those of the ILP-based algorithm, whereas the FF algorithm yields significantly higher costs. More precisely, the costs achieved by SODA are on average 2.19% higher than the costs achieved by the ILP-based algorithm; the costs of FF are 29.32% higher than those of ILP.

Fig. 5.7(b) shows the execution time of the three algorithms in each step. The time is shown in milliseconds, using logarithmic scale. The execution time of the ILP-based algorithm is consistently significantly higher than the execution time of the two heuristics. More precisely, the average execution time of the
ILP-based algorithm is roughly 26 seconds, while the average execution time is only about 36 milliseconds for SODA and 1 millisecond for FF. Moreover, in 8 cases, the execution time of the ILP-based algorithm reaches the timeout threshold of 60 seconds. In the case of a timeout, the ILP solver returns the best solution found so far as well as a lower bound on the optimum. In these cases, the output of the ILP-based algorithm may not be optimal; however, based on the lower bound on the optimum, we can establish that the cost of the deployment resulting from the ILP-based algorithm is at most 0.82% higher than the optimum.

Consistently across all experiments, we made the following observations:

- The costs of the deployment returned by SODA are only slightly higher than those of the ILP-based algorithm. On average across all runs in the experiments, the costs of SODA are 2.7% higher than those of ILP. The costs of FF are much higher, on average 88.8% higher than those of ILP.

- FF is very fast, taking on average just 0.6 millisecond per run. SODA takes longer, but is still very fast: its execution was on average 40.6 milliseconds per run; moreover, in each run, its execution time was below 300 milliseconds. ILP was on average more than 500 times slower than SODA, as it took on average almost 22 seconds per run, also hitting the 60 seconds timeout in many cases.

Fig. 5.8 shows how the three algorithms realize different trade-offs between solution costs and execution time. In particular, it can clearly be seen that SODA is almost as good as ILP in terms of solution costs, while being almost as fast as FF (note the logarithmic scale of the vertical axis). We can conclude that SODA is very fast and consistently leads to near-optimal results.

A further problem with the ILP algorithm is that its execution time exhibits large variance. Specifically, the relative standard deviation (i.e., ratio of standard deviation to mean) of the execution time of the ILP-based algorithm across all runs was 1.09, whereas the same metric is 0.86 for SODA.
6 Outlook: Ensuring data protection in fog computing

Fog computing is the natural next step in the evolution of cloud computing, bringing cloud-like elastic compute capacity to the network edge, near to end user devices [70]. This way, computation-sensitive tasks can be offloaded from the end devices (like mobile phones, wearable devices, or cameras) to fog resources (i.e., compute resources at or near the network edge, e.g., in routers, base stations, or geographically distributed data centers of telecommunication providers). Offloading is advantageous for many applications that require higher computational capacity than what is available in end devices. Compared to offloading compute tasks to a large centralized cloud data center, fog computing has the advantage of considerably lower latency in the data transfers, which is essential for several time-critical applications [55].

Nevertheless, fog computing is also subject to several challenges. In particular, fog computing offers a plethora of opportunities for malicious parties to gain access to, or even manipulate, sensitive information [59]. Some of these threats are inherited from cloud computing, but some are new and specific to fog computing. More importantly, concerns about data protection can significantly hinder the adoption of the fog computing paradigm.

Of course, there are several known security techniques with which the access to sensitive data can be protected. However, the available techniques also have some limitations (e.g., some assume the availability of special hardware) or drawbacks (e.g., overhead). Therefore we argue that security techniques should be applied in an adaptive way. That is, the most appropriate technique should be selected based on the current situation. Adaptations should be carried out at run time, since also the situation may change dynamically at run time. Therefore, the current system state has to be monitored, so that risks concerning data protection can be identified and mitigated on the fly.

6.1 Data protection challenges in fog computing

For a more detailed survey of the general field of security and privacy in fog computing, the reader is referred to [71]. Here we only review the most important challenges related to the protection of sensitive information in fog computing.

Just like in cloud computing, users lose control of their data by uploading them to a server that is beyond their control [3]. Thus, the provider operating the given cloud or fog resource may get access to users’ confidential data. The provider may also let third parties access the data – intentionally or unintentionally, with or without consent from the user – so that also these third parties may abuse the data. Moreover, because of the intrinsic multi-tenancy of both the cloud computing and fog computing paradigms, other users may also use the same server or fog resource, which might make it possible for those other users to gain unauthorized access to sensitive data. In some cases, it is also possible that users try to get access to confidential data of the provider, for instance to get to know important business secrets about the provider’s infrastructure. All these types of attacks are conceivable in both cloud and fog computing.

In addition, there are some specific characteristics of fog computing that make data protection even more challenging than in cloud computing:

- **Reduced physical protection.** While cloud data centers are typically protected by strict physical access control mechanisms (e.g., doors that can be opened only by authorized personnel with their entry cards), fog resources are often deployed “in the wild” where malicious parties can get physical access much more easily. Even more importantly, fog computing is mostly based on wireless networking technologies which may be broken into without physical contact. In contrast, cloud computing is mostly based on wired networks, which are easier to protect.

- **Less clarity about stakeholders.** In cloud computing, users choose service providers explicitly and deliberately, also giving explicit consent regarding the use of their data. On the other hand, in some
fog computing scenarios, a device may use a variety of fog services for offloading computations, without the user of the device – or the data subject about whom the device is collecting data – being aware of the stakeholders that operate those resources or have otherwise access to the resources.

- **Direct access to confidential information.** A device using fog computing resources may leak sensitive personal information even without transferring any data explicitly to the fog resources. For example, since devices prefer to use nearby fog resources, an attacker might be able to determine a user’s approximate location or the route of a mobile user based on which fog resources the user’s device has connected to, thus violating location privacy [51, 24]. Another example is the violation of usage privacy in smart metering where the information gathered by smart meters reveals usage patterns of electronic devices in a household [49].

- **Scarce resources.** Existing methods for data protection, such as advanced cryptographic protocols or data obfuscation techniques, are often resource-intensive. This, however, is a problem in end devices that have limited computational capacity, limited battery power, and limited network bandwidth.

For these reasons, data protection is a very challenging problem in fog computing.

### 6.2 The case for adaptive data protection

Fog computing systems are very dynamic: end devices connect to fog resources and then disconnect, fog resources appear and disappear, wireless network connections get stronger or weaker etc. [2, 17]. With all those changes, risk levels also keep changing. For instance, from the point of view of an end device, risk levels may be low if the device can connect to a known and trusted fog resource, but the risk of data protection issues gets much higher if the connection to the trusted fog resource is lost and an unknown fog resource must be connected instead.

As already mentioned, existing security techniques that can ensure the protection of sensitive data often have downsides. For instance, some encryption techniques introduce a significant performance overhead [47]. Another option is the use of secure hardware enclaves, i.e., special hardware enabling the protection of code and data even from attackers with highest operating system privileges [14]. Performing computations in a secure enclave thus shields the data both from co-located applications and even from the operator of the server. However, also the use of secure enclaves incurs some overhead and even more importantly, it presumes the availability of appropriate hardware.

For these reasons, we argue that data protection mechanisms should be applied in an adaptive manner. In other words, data protection mechanisms should be activated only when needed to minimize their negative impact on resource consumption; moreover, from available alternative mechanisms always the most appropriate one should be chosen, taking into account the sensitivity of the data and the current configuration of the fog system.

In the following subsections, we review how adaptive application of data protection mechanisms can be achieved – first in general, and then focusing on the viewpoints of end users and fog service providers, respectively.

#### 6.2.1 Enabling adaptations

The fundamental model underlying most adaptive systems is a control loop according to the MAPE (monitor, analyze, plan, execute) principle [30]. This principle can also be applied to the problem of adaptive data protection in fog computing, as follows.

The basis for any decision-making is monitoring. That is, the current configuration of the fog computing system needs to be monitored, including the available resources, the planned computations, the involved data, and any other information that may have an impact on risks (e.g., known vulnerabilities or reputation...
of stakeholders). Monitoring may be complemented by prediction, e.g., to predict the future availability of wireless network connections or the duration of offloaded tasks [44]. Based on the information provided by monitoring and potentially prediction, an analysis has to be carried out to determine the risks of data protection violation and the possible risk-mitigating actions. The results of the analysis form the input to planning. The aim of planning is to decide which risk-mitigating action(s) to take, based on the impact of the possible actions on both data protection risks and other system goals like performance or costs. Finally, the chosen actions have to be executed.

For implementing the MAPE loop, a model-based approach is advantageous. This means that a model of the fog computing system is maintained at runtime in a machine-readable format. Monitoring updates the model so that it remains in sync with reality. Analysis and planning can be performed directly on the model, while execution ensures that modifications performed on the model are also transferred to the real world.

We have already proposed a run-time model for reasoning about data protection in cloud systems. This model could be extended and adjusted to capture the necessary entities of fog computing.

### 6.2.2 Adaptation in end devices

In the simplest case, an end device wants to offload some computations to a fog resource. Monitoring and analysis could be used to answer the following questions:

- Can the provider of the fog resource be trusted (e.g., because of previous experience or because of high reputation in publicly available provider evaluations)?
- What security capabilities does the resource offer (e.g., secure hardware enclaves)?
- How sensitive are the data involved in the planned compute task?
- What would be the impact of the available client-side data protection mechanisms, and how critical would that impact be in terms of system goals like performance and energy consumption?

Based on these pieces of information, a sound decision can be made on the action to be taken. For example, if the data are not sensitive and/or the provider is trusted, then the computation can be offloaded without using further data protection mechanisms. Otherwise, if the targeted fog resource features secure hardware enclaves, then the computation should be performed within an enclave. Otherwise, if the resource situation allows it, an appropriate encryption technique should be used. If none of the above is applicable, then the computation should not be offloaded because the associated risks cannot be effectively mitigated.

There can also be more complicated cases, e.g., if not only an edge device and a fog resource are involved, but additionally a central cloud as well. To keep the analysis and planning manageable, the risk-pattern-based approach described earlier could be applied.

### 6.2.3 Adaptation in fog resources

For a provider of fog resources, the goal is to fulfill the data protection requirements, while serving as many end client devices as possible and also ensuring a smooth and efficient operation [64]. This leads to interesting optimization problems [37]. For example, if a subset of the resources owned by the provider offer secure hardware enclaves, then taking this into account when allocating client requests to resources is a useful lever for keeping costs low while fulfilling data protection requirements. Our previous experience has shown that, with appropriate optimization algorithms, if only a small fraction of resources offer secure hardware enclaves, this can already lead to considerable savings in energy consumption [40].
6.3 Research challenges

Above, we sketched why adaptive data protection in fog computing is sensible and how it might be achieved. However, to make adaptive data protection in fog computing a reality, several research challenges need to be addressed (the list is not intended to be exhaustive):

- Appropriate models should be devised that incorporate all entities of fog computing deployments (along with their attributes and relations) that are relevant for data protection.
- The underlying trust models and attack models need to be better understood, categorized, formalized, and made available to automated run-time decision-making.
- A catalog of risk patterns needs to be elaborated that capture the relevant risks to data protection in fog computing.
- Appropriate monitoring techniques are necessary to efficiently and effectively monitor fog computing systems.
- Analysis, planning, and optimization algorithms need to be elaborated that can efficiently cope with decision-making on the different layers of fog computing systems (end devices, fog resources, cloud).
- Algorithm efficiency and scalability are crucial to ensure that adaptation works well in real time even under high dynamics.
- Testing, auditing, and verification techniques need to be devised to improve the credibility of data protection solutions in fog computing.
- The interplay among multiple autonomous systems that perform self-adaptations on their own needs to be better understood, including the possible coordination models and emergent behavior.

Moreover, it would be advantageous for fog computing research in general to define and make publicly available some representative examples that can be used to objectively assess and compare different approaches.
Bibliography


[34] Philipp Leitner and Jrgen Cito. Patterns in the chaos – a study of performance variation and predictability in public FaaS clouds. *ACM Transactions on Internet Technology*, 16(3), 2016.


A Publications

In the following, three publications are attached that describe in more detail the approaches presented in this deliverable. The three publications are:


Finding risk patterns in cloud system models

Florian Kunz and Zoltán Ádám Mann
paluno – The Ruhr Institute for Software Technology, University of Duisburg-Essen, Essen, Germany

Abstract—The risk of unauthorized access to confidential data is a major problem in cloud computing. In previous work, the notion of risk patterns was introduced to capture configurations of cloud systems prone to data protection issues. In this paper, we devise a program for the automatic detection of risk patterns in cloud system models. Our program makes use of the Eclipse Modeling Framework, the model transformation library Henshin, and the modeling workbench Sirius to (i) enable security experts to describe cloud risk patterns in a compact way, (ii) enable the efficient automatic detection of risk patterns in the model of a cloud system, and (iii) support cloud experts in experimenting with the security implications of different cloud configurations. A case study and experiments demonstrate the applicability and scalability of the proposed approach.

Index Terms—cloud computing; security; risk management; data protection; graph pattern matching; run-time model

I. INTRODUCTION

Using cloud services usually involves storing and processing data in the cloud. Some of those data may be confidential: for example, personal data that must be protected according to the EU General Data Protection Regulation (GDPR [6]), or trade secrets that must be protected from competitors. Protecting confidential data in the cloud is very challenging: customers of cloud services lose control of their data when the data are uploaded to the cloud, and numerous other parties – e.g., the service provider and other tenants – may succeed in obtaining unauthorized access to the data [16], [8].

Protecting data in the cloud is made difficult by the complexity of cloud systems, which consist of many hardware and software components [7]. These components are accessible by different stakeholders and host different pieces of data. A malicious attacker having access to one component may be able to find a path of components through which it can get access to confidential data. For instance, an Infrastructure-as-a-Service (IaaS) provider may abuse its access to physical servers to get control of applications running in virtual machines hosted on those servers, and then use such an application to access confidential data stored in a remote database. If such an attack is anticipated, security engineers can protect against it by using appropriate security controls (e.g., access control or encryption). Each security control may block some potential attack paths. However, the higher the complexity of a cloud system, the more difficult it becomes to verify that every potential attack path has been blocked. Thus, it is difficult to assess the risk that certain types of attacks would be successful in gaining unauthorized access to confidential data.

A further issue that complicates data protection is the dynamic nature of cloud systems. Workloads change, prices change, servers fail, new servers become available, software is upgraded etc., leading to frequent changes in the cloud. All changes may impact the risk of data protection violations: a change may eliminate a vulnerability, or may introduce a new one. Hence, assessing data protection risks of a cloud system is not a one-off activity, but must be carried out continually. Moreover, it has to be automated to enable the quick detection of newly arising data protection risks during system operation.

A promising approach for automatically detecting data protection risks in cloud systems uses so-called risk patterns that capture sub-structures of cloud configurations associated with high data protection risks [13]. Moreover, a model of the cloud system is used and continually updated based on run-time monitoring of the system to ensure that the model is always in line with the real cloud system. [13] suggested that graph pattern matching algorithms could be used to automatically search for instances of the risk patterns in the model of the cloud system. However, the approach was not implemented, leaving several important questions open:

- Can graph pattern matching indeed be used for finding risky cloud configurations?
- Can risk patterns be captured in a concise format?
- Is the search for risk patterns fast enough for online use?

In this paper, we address these gaps. We provide an implementation of the risk pattern approach using the Eclipse Modeling Framework (EMF), the model transformation library Henshin, and the modeling workbench Sirius. We show that:

- Graph pattern matching is not enough to capture all relevant information. However, using more powerful EMF-based model matching, we were able to implement the automatic search for risk patterns.
- Risk patterns can be concisely captured in the form of Henshin rules, which are both human-readable and machine-readable.
- The search for risk patterns is quite fast: for cloud models of up to 17,000 nodes, the search took less than 0.4 seconds on a commodity computer.

Therefore, our implementation yields significant new insights regarding the applicability of the risk pattern approach.

II. PRELIMINARIES

In this section, we summarize information about the risk pattern approach suggested in [13], [9]. The aim of the approach is to detect cloud configurations that are associated with an unacceptably high risk of data protection violations. To achieve this aim, a model-based approach is suggested. A cloud system model represents all entities that may be relevant for data protection, including infrastructure elements like physical machines (PM) and virtual machines (VM), middleware...
elements like application servers, application components, databases, and stakeholders [9]. It is assumed that run-time monitoring of the cloud continually updates the model so that it always is in line with the actual cloud configuration.

Beside the cloud system model, the other key artefact is a set of risk patterns. A risk pattern captures a problematic sub-structure of a cloud configuration which, if contained in the cloud system model, would lead to unacceptably high data protection risks. Risk patterns are model fragments using the same entities as the cloud system model, i.e., they are based on the same meta-model, but have different semantics. While the cloud system model describes a specific cloud configuration, a risk pattern captures a problematic situation that may or may not be found in the models of different cloud configurations.

A risk pattern specifies model elements (entities, relations, attribute values) that must be present in a model to match the given pattern, as well as model elements that must not be present in a match. For elements not specified in the risk pattern, their presence does not matter.

In [13], two sample risk patterns were presented, see Fig. 1. The first risk pattern describes a situation in which sensitive data of a data subject are stored in unencrypted form in a database which is operated by a Platform-as-a-Service (PaaS) provider, and the provider is not trusted by the data subject. This is a high data protection risk since the untrusted provider may gain unauthorized access to the sensitive data. The risk labeled “No trust” is an example of the presence of a model element being prescribed by the risk pattern, i.e., this risk pattern will only match cloud models in which there is no trust relation between the data subject and the PaaS provider. The second risk pattern captures the situation in which sensitive data about an EU citizen are processed by an application component hosted outside the EU. This is problematic because the GDPR stipulates that personal data should only be processed within the EU. The risk pattern captures the chain of relations from the data through the application component and a VM to the PM, the location of which causes the problem.

1Actually, the GDPR allows processing of personal data also in certain countries outside the EU. In this respect, Risk Model B of [13] is not fully accurate. The location attribute of the PM should be something like “prohibited country” instead of “non-EU”

In [13], the model of a real cloud system was used for validation (see Fig. 2). The two risk patterns were manually searched for in the model, and one instance of each of them was found. This result was in line with the evaluation by security experts.

The risk pattern approach is promising as it can capture complex configurations that lead to data protection risks. This can be seen on the two presented example risk patterns: the data protection issues arise when the depicted configurations, including the given entities, attributes, and relations, can be found in a cloud model. However, existing work on the risk pattern approach is on a high level of abstraction, in particular lacking an implementation of the pattern matching process.

III. IMPLEMENTATION

For implementing the risk pattern approach, we need more than just a graph pattern matching algorithm. As can be seen from Figures 1–2, we need the ability to reason about not only graph structures (vertices and edges), but also object types, attribute values, and relationship types. The types can also form an inheritance hierarchy. E.g., it was shown in [12] that it is useful to extend the meta-model with two new types AtomicComponent and CompoundComponent, both inheriting from Component. The Component element of Risk Pattern B in Fig. 1 should then match any object of type Component or of any of its sub-types.

Therefore, we decided to use the Eclipse Modeling Framework (EMF [15]). EMF is a widely used modeling framework based on the Eclipse environment and supported by a variety of tools. In particular, Henshin is a model transformation library, also supporting pattern matching in EMF models [3]. EMF also allows direct model manipulation with custom Java code.

Fig. 3 shows the interaction of the used tools (EMF, Henshin, Sirius, MatchFinder) and the artefacts created or used by the tools (meta-model, cloud system model, risk patterns). With the help of EMF, we created a meta-model of cloud configurations. This meta-model is stored in EMF's
n native format (Ecore). The risk patterns are modelled with the Henshin editor in the form of Henshin rules. Pattern matching and the handling of found matches are performed by the MatchFinder tool, which is a combination of custom Java code and the Henshin engine. To foster experimentation, visualization, and manual exploration of design alternatives during design time, we also created an editor with which the cloud system model can be displayed and edited. For this purpose, the Sirius plug-in was used [17].

a) Meta-model in Ecore: We translated the meta-model from [12] to Ecore format. This required some small technical changes, like the insertion of a root node. We also discovered some small inconsistencies between the meta-model of [12] and the cloud system model and risk patterns of [13]; e.g., the meta-model of [12] contained no relation between actors that would represent a trust relationship, which is needed in the example risk patterns of [13]. We fixed the inconsistencies by making appropriate changes to the meta-model, the cloud system model, or the risk patterns. Such inconsistencies can remain undiscovered in a manual validation such as in [13], [9], [12]; an implementation and validation like we perform here has a much higher chance of discovering such issues.

b) Risk patterns as Henshin rules: Henshin provides an editor for the creation of Henshin rules based on an Ecore model. A Henshin rule specifies a model transformation. Conceptually, a Henshin rule consists of two parts: the left-hand side (LHS) specifies a pattern to be found in the input model, while the right-hand side (RHS) specifies how the found pattern is to be changed in the output model. The Henshin editor presents the LHS and RHS together in an integrated object model, in which Henshin-specific annotations are used to mark the role of the model elements (objects, relations, attributes) with respect to the LHS and RHS. For specifying risk patterns, we use two of Henshin’s annotations: preserve and forbid. Every model element annotated with preserve has to be found in the input model for a match. Model elements marked with forbid must not be present for a match to be found. Figures 4–5 show the realization of the two example risk patterns from Fig. 1 as Henshin rules. As can be seen, the annotations preserve and forbid are used to mark model elements that must be present, respectively must not be present in a match.

c) Editor for cloud system models: Beside the risk patterns, a cloud system model is needed in which the risk patterns are searched for. The cloud system model can be a design-time artefact that a cloud security analyst works with to analyze different cloud configurations, or a runtime model updated through monitoring reflecting the current configuration of the cloud system. We created an editor for working with cloud system models, which may be used in the design-time setting by the cloud security analyst. In a run-time setting, this tool may be used to visualize the current system configuration to a human operator.

We used Sirius [17] to create a graphical editor, in which objects, their attributes and relationships can be created in accordance with the meta-model, displayed on a canvas, and edited. We constructed creators for the instantiable types of the meta-model, which are made available to users in the form of a toolkit. For the relations, providing a separate creator for each association in the meta-model would have led to a huge number of creators, resulting in decreased usability.
Hence we implemented a generic creator for relations, which automatically infers the type of the relation based on the types of the start and end nodes.

d) The control logic: MatchFinder is a Java project that integrates the other pieces. It uses the risk patterns in form of Henshin rules, the cloud system model created with the Sirius-based editor, and the meta-model in Ecore format. MatchFinder controls the pattern matching process and handles the results. The SearchInitiator within MatchFinder reads and converts all input models and initiates the pattern matching using the Henshin tool set. The result of the pattern matching is a (possibly empty) set of matches. For each match, MatchFinder displays a human-readable textual description of the match. The matches are also graphically shown within the cloud model in the Sirius-based editor.

IV. Validation

To validate our implementation, we used the same example risk patterns and cloud model as in [13], except for some small changes required to ensure consistency with the meta-model. Figures 4–5 show our realization of the example risk patterns as Henshin rules. We found it straight-forward to model risk patterns in the form of Henshin rules. Henshin rules provide a concise representation of risk patterns with well-defined syntax and semantics, which we found easy to understand.

Fig. 6 shows the realization of the example cloud system model with the help of our Sirius-based editor. The Sirius-based editor made it easy to create the cloud system model or to make changes to it.

After running MatchFinder, we can check both in the textual output and in the Sirius-based editor that exactly one instance of each of the two risk patterns was found, just like in [13]. Fig. 6 shows the output in the Sirius-based editor, where the objects of the cloud model participating in at least one found match are marked. It is important to note that, while Risk Pattern B contains an object of type Component (cf. Fig. 5), this matches the object of type AtomicComponent in the cloud model (cf. Fig. 6), since AtomicComponent is a sub-type of Component in the meta-model.

We also performed additional tests, where small changes were made to the cloud system model. In all these tests, all present instances of the risk patterns were found, as we verified manually. Furthermore, our validation study also showed that the manipulation of all important artefacts (meta-model, cloud system model, risk patterns) was simple and intuitive with the implemented toolset, suggesting that the used tools are appropriate and together form a good technical basis for the management of data protection risks in cloud systems.

V. Empirical Results

We performed controlled experiments to investigate the scalability of our approach. To generate larger models with a realistic structure, we enlarged the example cloud model from Section IV in a sequence of 10 steps. After each step we checked the duration needed by MatchFinder to search for (i) Risk Pattern A, (ii) Risk Pattern B, (iii) both risk patterns together. The enlarged models consist of several copies of the example model which share the same data subject node. The model of the first step consists of the basic example model with 18 nodes. Each further step adds the example model 100 times except the data subject node. Thus the model of the second step consists of $18 + 100 \cdot (18 - 1) = 1,718$ nodes and the model of the 10th step contains $18 + 100 \cdot (18 - 1) = 17,018$ nodes. The measurements were performed on a Surface Pro 3 laptop with Core i5-4300U CPU at 2900MHz, 4GB RAM, Windows 10 Professional, and Java 9.

Fig. 7 shows the duration of MatchFinder for the models of increasing size. As can be seen, the execution time scales roughly linearly with the size of the model. Looking for Risk Pattern A which consists of only 5 nodes needs less time than looking for Risk Pattern B which consists of 7 nodes. Looking for both risk patterns at the same time needs more time than looking for just one; however, if both risk patterns have to be looked for, then it is faster to look for them together than to look for them separately. This is probably due to the overhead of setting up the search itself, which is incurred only once if the two risk patterns are searched for together.

Most importantly, Fig. 7 shows that the duration is quite low even for the biggest models tested. In the first step, the search took 104 ms while the duration in the last step is 346 ms for

![Fig. 7. Duration of pattern matching for growing cloud system models](https://restassuredh2020.eu/)

Time [ms]

- Risk pattern A & B
- Risk pattern A
- Risk pattern B

Cloud system model size [number of nodes]
a model with over 17,000 objects. Thus the duration increases by around 3.3 times while the size of the model increases by around 945 times. These findings indicate that the presented implementation scales well even for big cloud systems.

We also investigated how execution time scales with the number of edges in the model. We took one of the models from the previous experiments and added random edges to it, thus keeping the number of nodes constant while increasing the number of edges. We repeated this experiment with several starting models of different size. Also, we considered two approaches for adding new edges: (i) only adding edges inside the copies of the original example model of 18 nodes, (ii) adding edges between copies of the original example model. Our experience from these experiments is that increasing the number of edges hardly influences the time needed for the pattern matching. The duration remains low even for the biggest models considered (more than 100,000 edges).

VI. RELATED WORK

A large body of research addresses specific vulnerabilities of cloud systems and specific techniques to prohibit attacks. Examples include access control [10], trust management [11], security certification [2], and privacy-preserving analytics [14]. The risk pattern approach considered in this paper is fundamentally different from these works, as it focuses on the detection of cloud configurations with high data protection risks in general, instead of specific security controls.

The risk pattern approach was suggested in [13] and extended with a meta-model for cloud system models in [9]. A catalog of risk patterns modeling known attacks was presented in [12]. This paper is the first that presents an implementation and thus validates the applicability of the approach.

SecVolution [5] is also a model-based approach for detecting system configurations that may violate some security requirements, also using EMF and Henshin. However, SecVolution is focused on software, whereas the risk pattern approach includes all relevant layers of a cloud system. The risk pattern approach allows more accurate reasoning about cloud vulnerabilities. SecVolution aims at supporting manual enhancement of software by semi-automatically changing design artefacts, while the risk pattern approach aims at the fully automatic and quick detection of risky configurations.

Some works addressed the management of security and privacy risks in cloud settings. E.g., [4] elaborated on the necessity of quantified risk assessment for cloud-based processes, but also on the difficulties and challenges of such assessment, and [1] assessed the risks associated with migrating a process to the cloud. These approaches are complementary to our approach, as they can be used for determining the cloud configurations that must be avoided, which can be captured in the form of risk patterns.

VII. CONCLUSIONS

This paper has provided an implementation of the risk pattern approach for automatically detecting forbidden cloud configurations, thus validating the applicability of the approach, and leading to new insights about the requirements on the used pattern matching algorithm. Risk patterns can be represented as Henshin rules in a compact manner which is both human-readable and machine-readable. The approach is fast for even large models, allowing it to be used automatically at run time. An important area for further research is the automatic mitigation of the found risks by appropriate adaptations of the cloud configuration.

Acknowledgments. This work was partially supported by the European Union’s Horizon 2020 research and innovation programme under grant 731678 (RestAssured).

REFERENCES

Resource optimization across the cloud stack

Published in IEEE Transactions on Parallel and Distributed Systems, 29(1):169-182, 2018

Zoltán Ádám Mann

Abstract—Previous work on optimizing resource provisioning in virtualized environments focused either on mapping virtual machines (VMs) to physical machines (PMs) or mapping application components to VMs. In this paper, we argue that these two optimization problems influence each other significantly and in a highly non-trivial way. We define a sophisticated problem formulation for the joint optimization of the two mappings, taking into account sizing aspects, colocation constraints, license costs, and hardware affinity relations. As demonstrated by the empirical evaluation on a real-world workload trace, the combined optimization leads to significantly better overall results than considering the two problems in isolation.

Index Terms—Virtual machines; VM placement; VM consolidation; VM selection; VM sizing; cloud computing; data center

1 INTRODUCTION

As cloud data centers (DCs) are serving an ever growing demand for computation, storage, and networking, their efficient operation has become a high priority. Cloud providers seek to serve as many customer requests as possible and to decrease operational costs. Operational costs are largely driven by electricity consumption, which also impacts the environment. At the same time, cloud providers must also fulfill service-level objectives (SLOs) on performance, availability, and security.

Virtualization has been widely adopted in DCs to consolidate workload on the necessary number of physical machines (PMs) with high utilization of the available hardware resources. For this purpose, virtual machines (VMs) are used as the virtual infrastructure for running the workload, enabling the isolated execution of multiple applications on the same PM. However, virtualization also has some drawbacks (e.g., overhead [30]) and limitations (e.g., no perfect isolation of colocated VMs from each other [7], [27]).

Because of its impact on costs, application performance, SLOs, and the environment, optimization relating to the management of VMs has received considerable attention in the last couple of years. As shown in our recent survey [30], most previous research efforts fall into one of two categories: VM placement and VM selection. VM placement is a problem faced by Infrastructure-as-a-Service (IaaS) providers: how to determine a mapping of VMs to PMs with the main objective of minimizing overall energy consumption. On the other hand, VM selection is faced by IaaS tenants concerned with assigning application components1 to VMs.

The two problems are quite different: VM placement is about physical resources, their utilization and power consumption, whereas VM selection is concerned with lease costs and application-level performance metrics. The central notion that connects the two perspectives is the VM.

Although VMs play an important role, especially in a public IaaS setting, we argue that VMs are just a tool for mapping tenants’ application components to the provider’s PMs in a safe and manageable fashion. Tenants’ main objective is to find hosts for their applications, providers’ objective is to utilize their infrastructure by accommodating workload that is valuable for their clients, and thus realize revenue. VMs can be seen as wrappers around application components that make all this possible in a manageable way. In this respect, VM placement and VM selection are just two sides of the same coin. Most importantly, the two problems influence each other.

A simplified example is shown in Fig. 1. Here, we consider a single resource dimension (e.g., only CPU) and assume that all PMs have the same capacity according to this resource. The capacity of the PMs is taken to be 1. We consider six components with resource need 0.3 each (i.e., each component requires 30% of the capacity of a PM). Further, we assume that a VM adds an overhead of 0.05 to the size of the contained component(s) in terms of resource consumption. The three subfigures show the effect of different VM selection policies on VM placement. In Fig. 1(a), the VM selection policy selects a dedicated VM for each component, resulting in 6 VMs of size 0.35 each, the placement of which requires at least 3 PMs. In Fig. 1(b), components are grouped pairwise into VMs, resulting in 3 VMs of size 0.65 each, the placement of which again requires 3 PMs. In Fig. 1(c), groups of 3 components are mapped to VMs, resulting in 2 VMs of size 0.95 each, and these can be hosted by 2 PMs. Therefore, this third scenario leads to approximately 33% energy savings. However, if we continue this line of thought and map 4 components into a single VM, this would result in VMs of size 1.25, which cannot be accommodated by the available PMs without severe resource overload.

As demonstrated by this example, VM selection influ-

1. In this paper, the term “component” denotes a component of an application, i.e., a software component.
ence VM placement in a non-trivial way. Therefore we argue that, at least in a private cloud setting, where VM selection and VM placement are in the hands of the same organization, the two kinds of optimization should be carried out in a closely coupled way. So, the setting of this paper is the IT department of an organization running a private cloud, in which both VM selection and VM placement are carried out by the IT department. The main questions addressed by this paper are:

- How much can we gain by optimizing VM selection and VM placement together, in a joint optimization?
- If the two problems are solved separately, how much can we gain by incorporating knowledge about VM placement into VM selection and vice versa, by incorporating knowledge about VM selection into VM placement?

To answer these questions, we compare several algorithms, from an integrated approach for the joint selection-and-placement problem to complete separation of the two problems. Also some approaches are investigated that are in between, meaning that they solve the two problems separately but include some information about one of the problems into the solution of the other. To compare the algorithms, we use several metrics including energy consumption, license costs, compliance with hardware affinity constraints, and compliance with colocations constraints of the resulting system configuration.

Next, related work is reviewed in Section 2, followed by the problem formalization in Section 3 and different possible algorithms in Section 4. Empirical experience with applying the presented algorithms to real-world workload data is presented in Section 5 and Section 6 concludes the paper. The online supplemental material contains a more detailed description of the aspects that VM selection and placement need to account for.

## 2 Previous Work

As shown in our recent survey [30], most previous research efforts on VM mapping problems fall into one of two categories: VM placement is concerned with mapping VMs to PMs in a DC, while VM selection considers the problem of mapping application components to VMs. In the taxonomy introduced in [32], the first one is the Single-DC problem, while the latter is the Multi-IaaS problem (mirroring the fact that cloud users can choose among a number of IaaS offerings).

### 2.1 VM placement

Even within the Single-DC problem, many different problem variants have been considered. The most important differentiating factors are:

- The set of resource types considered:
  - Many papers consider only the CPU [4]–[6], [9], [11], [20], [24].
  - Other papers included, beside the CPU, also some other resources like memory, I/O, storage, or network bandwidth [3], [8], [18], [34], [44].

- The considered cost factors:
  - Many papers focus on the number of active PMs because it largely determines the total energy consumption [4], [11], [39].
  - Some also take into account the load-dependent dynamic power consumption of PMs [1], [5], [15], [18], [20], [43], [47].
  - A further objective of some papers is to minimize the number of overloaded PMs because of the performance degradation that results from overloads [5], [11], [44].
  - Some papers also considered the cost of migration of VMs [5], [11], [37], [43].

As noticed by several researchers, the special case of the Single-DC problem in which a single resource type is considered and the only objective is to minimize the number of used PMs is equivalent to the well-known bin-packing problem. On one hand, this means that the Single-DC problem is strongly NP-hard so that the existence of an efficient exact algorithm is very unlikely. On the other hand, simple packing heuristics like First-Fit (FF), Best-Fit (BF), and First-Fit-Decreasing (FFD) are known to perform very well on bin-packing. Hence, several papers proposed to adopt such heuristics to the VM placement problem [4], [5], [20], [28], [44].

### 2.2 VM selection

Concerning VM selection (the Multi-IaaS problem), also many different problem formulations have been suggested.
Similarly to the Single-DC problem, most works focus on computational power [12], [29], [46] but a few works also consider other resource types like memory [25], [26], [35]. The main optimization objective is to find the best trade-off between performance and VM lease costs, which typically means that either the minimum required performance is given and costs must be minimized or the acceptable costs are constrained and performance must be maximized. Performance is often defined in terms of the makespan, i.e., the time between starting the first task and finishing the last one, in some cases also allowing dependencies among the tasks [10], [19], [21], [35].

Several different models have been investigated also in terms of VM lease costs. Most works consider costs proportional to VM usage time [9], [12], [22], [29], [45], [46], but some also add fees depending on consumed resource usage [26], [35] or discounts for long-term VM rental [19], [26]. Spot instances have also been considered [14].

Another relevant topic is auto-scaling, aiming to determine the number of necessary instances of a given VM to serve the current load [2], [40]. This can also be seen as a kind of VM selection problem.

### 2.3 Interplay of VM placement and VM selection

The papers cited above address either VM placement or VM selection in isolation. Although both problems have received much attention, their inter-dependence has hardly been studied. We are aware of only two papers by other researchers that made first steps into this direction. One of them is the recent work of Piraghaj et al. [36]. The focus of that paper is on selecting optimal VM sizes based on the characteristics of the tasks to be allocated. The objective is to reduce energy consumption by minimizing resource wastage. Each VM is assumed to have a fixed size irrespective of its workload, and the difference between the VM’s size and the total size of its workload is wasted.

In contrast, this paper assumes that a VM’s real size (as taken into account by the provider in VM placement decisions) follows the capacity requirements of its workload. The rationale is that resource usage is most of the time significantly below the peak, yielding a great opportunity for DC operators to consolidate VMs based on their current load and continuously adapt the placement accordingly, always using just the necessary number of active PMs [43]. Another important difference is that the work of Piraghaj et al. [36] did not consider migrations, whereas we do. Through these differences we believe to have a more realistic model, in which the sought trade-offs and the objectives are also somewhat different (opportunities for consolidation through migration versus minimization of wastage through sizing).

The other relevant paper is due to Ganesan et al. [17]. That work is in the context of a Software-as-a-Service provider that wants to allocate the components of its applications to VMs. The focus of the work is on VM sizing, namely, determining the dedicated and shared capacity for the VMs, based on past observations of the applications’ workload. Their algorithm also outputs recommendations for VM placement, like which VMs can be placed statically and which ones need dynamic placement. However, the actual allocation of VMs to PMs is not carried out; they assume that it is done by some external algorithm. In contrast, we are interested in the impact of selection on placement; it is unfortunately not possible to tell how good that approach is in this respect. Another limitation of that paper is the assumption that each application component is mapped to a separate VM, whereas we also allow to co-locate multiple components in the same VM.

In our own previous work, we have started investigating the connections between VM selection and VM placement [33]. In particular, we compared three different VM selection algorithms in combination with the same VM placement algorithm; our results suggested that the more information the VM selection algorithm has about the PMs, the current VM placement, and the VM placement algorithm, the better overall results can be achieved. In that work, components and VMs were only characterized by their size; in contrast, this work analyzes a similar question in the context of a much more general problem formulation, featuring beyond the mere size of the components also license costs, colocation constraints and hardware affinity constraints. Therefore we believe that the results of this paper are more relevant for practical use.

### 3 Problem Formulation

Based on the analysis of the relevant aspects of the problem (details can be found in the online supplemental material), we came to the problem model summarized in Figure 2.

The problem model revolves around components that are deployed in VMs, which in turn are deployed in PMs. Let $C$, $V$, and $P$ denote the set of components, VMs, and PMs, respectively. For a component $c \in C$, $v(c)$ denotes the VM where $c$ is deployed; likewise for a VM $v \in V$, $p(v)$ denotes its hosting PM.

The size of a component encodes its resource requirements along multiple resource types as a $d$-dimensional vector.

2. The numbers near the ends of links mean minimum and maximum cardinalities and * means infinity. For instance, the numbers near the link between Component and VM mean that a component is deployed in exactly one VM, whereas a VM can host an arbitrary number (from 0 to infinity) of components.
vector. Here, \( d \) is the number of considered resource types, e.g., if CPU and memory are considered, then \( d = 2 \). The size of a VM is also a \( d \)-dimensional vector: the sum of the sizes of the components deployed in the given VM, plus the overhead of virtualization (the size vector of an empty VM).

For a VM \( v \in V \), its size thus is computed as

\[
size(v) = s_0 + \sum_{c \in C : v(c) = v} size(c),
\]

where \( s_0 \in \mathbb{R}^d \) is the size vector of an empty VM.

Each PM \( p \in P \) has given capacity according to each of the considered resource types. Therefore, the capacity of a PM \( p \) is given by a \( d \)-dimensional vector \( \text{cap}(p) \). The mapping of VMs on PMs must respect the capacity of the PMs, as captured by the following constraint:

\[
\forall p \in P : \text{load}(p) = \sum_{v \in V : p(v) = p} size(v) \leq \text{cap}(p).
\]

Note that here, \( \leq \) is a component-wise comparison of \( d \)-dimensional vectors: for \( x, y \in \mathbb{R}^d \), \( x \leq y \) if and only if \( x_j \leq y_j \) for each \( j = 1, \ldots, d \).

The state of a PM can be either on or off. The operating system of VM \( v \) is \( \text{os}(v) \). For each component \( c \), the list of operating systems is given on which it can run; \( \text{os}(v(c)) \) must be an element of this list.

A colocation constraint relates to a pair of components. The type of the constraint can be one of must, should, should not, and must not. Moreover, it is given for each colocation constraint whether it relates to the colocation in the same VM or the same PM. With this mechanism, we can model all colocation aspects described in the online supplemental material. For instance, shared-memory communication between components leads to a must-constraint on VM level, meaning that they must be in the same VM, whereas intensive but loosely-coupled communication may lead to a should-constraint on PM level, meaning that they should be in the same PM. Security concerns may necessitate a must-not-constraint on PM level, meaning that they must not be in the same PM etc.

A component may have a license assigned to it, if it is placement-relevant because the license fee is proportional to either the number of VMs or the number of PMs running components with the given license. For a VM-based license \( \ell \), let \( V(\ell) \) denote the set of VMs containing at least one component associated with license \( \ell \), then the license fee to be paid because of \( \ell \) is \( f\text{ve}(\ell) \cdot |V(\ell)| \). Similarly, if \( \ell \) is a PM-based license and \( P(\ell) \) denotes the set of PMs containing at least one component associated with license \( \ell \), then the license fee to be paid because of \( \ell \) is \( f\text{ve}(\ell) \cdot |P(\ell)| \). The total license fee to be paid is the sum of the fees for each license.

As a consequence, if multiple VMs containing components with the same license \( \ell \) are in the same PM, then

- the license fee has to be paid only once if \( \ell \) is a PM-based license;
- it has to be paid for each VM if \( \ell \) is a VM-based license.

A PM may possess some PM features. A hardware (HW) affinity constraint can specify the relation of a component to a PM feature. The type of the HW affinity can be either must (the component definitely requires a PM with the given feature) or should (the component benefits from a PM with the given feature).

The power consumption of a PM is a function of its CPU load. As in several previous works [4], [18], [43], we use a linear approximation, i.e., the power consumption of a PM with CPU capacity \( c \) and CPU load \( x \) is given by

\[
W(x) = W_{\text{min}} + (W_{\text{max}} - W_{\text{min}}) \cdot x/c,
\]

where \( W_{\text{min}} \) and \( W_{\text{max}} \) are the minimum and maximum power consumption of the PM, respectively. Table 1 gives an overview of the used notation.

Now we summarize the problem’s inputs, outputs, constraints, and objectives. The inputs are:

- the set of components with the associated colocation constraints, licenses, and HW affinities;
- the set of PMs with their PM features.

The output consists of

- the set of VMs to be used,
- the mapping of components to VMs,
- and the mapping of VMs to PMs.

The solution must respect the PM capacity constraints, the requirements of the components in terms of VM OS, the colocation constraints of type „must” and „must not,” and the HW affinities of type „must.” There are multiple objectives: minimizing the total energy consumption of the PMs, minimizing the total license fee, maximizing the number of satisfied colocation constraints of type „should” and „should not,” and maximizing the number of satisfied HW affinities of type „should.”

As with any model, this problem formulation also abstracts from some technical details. For example, the transient processes of turning a PM on or off, deploying a VM on a PM, or deploying a component in a VM are not considered, nor their overhead in terms of time and energy. This is in line with the problem formulations used in most previous works in this area [30] and represents a good approximation for long-running software components. For very dynamic settings, the problem formulation – and the subsequent algorithms – may need to be extended in this respect.

4 Mapping algorithms

In this section, we first devise an algorithm for the joint VM selection and VM placement problem, called COMBINED.
Then, for the purpose of comparison, we also introduce some algorithms for only VM selection respectively only VM placement.

VM placement and VM selection are tough combinatorial problems for which optimal methods are unfortunately intractable for practical problem sizes [31]. Therefore, in line with most existing works, the algorithms presented here are all heuristics.

4.1 Combined VM selection and VM placement

The aim of the COMBINED algorithm is to map a new component $c$ on a VM and a PM. This can be done in several ways:

- Starting a new VM $v$ to host $c$ and placing $v$ on a PM
- Selecting an existing VM $v$ to host $c$ and keeping $v$ on the PM where it is
- Selecting an existing VM $v$ to host $c$ and migrating $v$ to another PM

These three principal ways of mapping can be unified by considering all VMs from $V$ \cup \{v^*\} as possible hosts for $c$, where $V$ is the set of existing VMs and $v^*$ is a new VM, and considering all PMs as possible hosts for the selected VM. The basic idea of the COMBINED algorithm is to examine all these possibilities and choose the best one.

One challenge that needs to be tackled is the potentially large number of possible configurations to examine, namely $(|V| + 1) \cdot |P|$. The naive approach of examining all possible configurations can be rather time-consuming if $|V|$ and $|P|$ are large, taking also into account that examining a possible configuration in terms of several objectives is also non-trivial. Note that this would still be a polynomial-time algorithm, but in order to quickly react to tenants’ deployment requests, the selection and placement algorithm has to be fast even for large DCs.

For this reason, we decided to first filter the set of candidate PMs and VMs and take only the promising ones into account. As shown in Algorithm 1, we collect the promising VMs in a set $V^*$ and the promising PMs in a set $P^*$. We start by placing a new VM with an appropriate OS in $V^*$ (line 3). If there is a colocation constraint of type must or should between $c$ and another component $c'$, then either the VM or the PM hosting $c'$ is also added, depending on whether it is a VM-level or PM-level colocation constraint (lines 4-12). Such VMs/PMs are indeed promising candidates, since mapping $c$ onto them would satisfy the colocation constraint. Similarly, if $c$ has a VM-based license, then all VMs containing a component with the same license are added to $V^*$ (lines 13-18), whereas if $c$ has a PM-based license, then all PMs containing a component with the same license are added to $P^*$ (lines 19-26). These are again promising since mapping $c$ onto them would incur no license fee. For hardware affinity constraints of $c$, all PMs offering the needed feature are added to $P^*$ (lines 27-32). From all PMs where $c$ would fit without overload, the ones with the highest load are also added to $P^*$ (lines 33-35), since mapping $c$ onto them would lead to good utilization and thus to relatively low energy consumption. In all these steps, if a PM $p$ is added to $P^*$, then the VMs hosted on $p$ are added to $V^*$. Finally, we add some further random VMs and PMs to $V^*$ and $P^*$ to extend the search space (lines 36-38). This is important especially if there are few VMs or few dependencies (colocation constraints, common licenses, affinity constraints). At the end, $V^* \times P^*$ defines the set of candidates to examine.

### Algorithm 1 Determining candidate PMs and VMs

1: procedure CANDIDATES($c$)
2: $V^*$ ← $\emptyset$, $P^*$ ← $\emptyset$
3: Add to $V^*$ a new VM with OS compatible with $c$
4: for all must or should colocation constraint of $c$ do
5: Let $c'$ be the other component of the constraint
6: if VM-level colocation constraint then
7: Add $v(c')$ to $V^*$
8: else PM-level colocation constraint */
9: Add $p(v(c'))$ to $P^*$
10: Add each VM on $p(v(c'))$ to $V^*$
11: end if
12: end for
13: if $c$ has a VM-based license $\ell$ then
14: for all $v \in V$ do
15: if there is a component in $v$ with license $\ell$ then
16: Add $v$ to $V^*$
17: end if
18: end for
19: else if $c$ has a PM-based license $\ell$ then
20: for all $p \in P$ do
21: if there is a component in $p$ with license $\ell$ then
22: Add $p$ to $P^*$
23: Add each VM on $p$ to $V^*$
24: end if
25: end for
26: end if
27: for all hardware affinity constraint of $c$ do
28: for all PM $p$ with the given PM feature do
29: Add $p$ to $P^*$
30: Add each VM on $p$ to $V^*$
31: end for
32: end for
33: Let $P''$ be the set of PMs on which $c$ would fit
34: Sort $P''$ in decreasing order of CPU load
35: Add the first $k_1$ PMs of $P''$ to $P^*$ and their VMs to $V^*$
36: Add $k_2$ random VMs to $V^*$
37: Add $k_3$ random PMs that are on to $P^*$
38: Add $k_4$ random PMs that are off to $P''$
39: return $(V^*, P^*)$
40: end procedure

**Fig. 3. Minimization objectives of VM selection and placement**

<table>
<thead>
<tr>
<th>Objectives</th>
<th>Hard factors</th>
<th>Soft factors</th>
<th>License cost</th>
<th>Power consumption</th>
<th>Nr. of violated must colocation constraints</th>
<th>Nr. of violated must not colocation constraints</th>
<th>Nr. of violated hardware affinities</th>
<th>Nr. of PM overloads</th>
<th>Nr. of violated should colocation constraints</th>
<th>Nr. of violated should not colocation constraints</th>
<th>Nr. of violated hardware affinities</th>
<th>VM size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RestAssured Consortium

https://restassuredh2020.eu/

D5.3 – October 31, 2019 81 of 112
The other challenge in devising the combined algorithm is rooted in the multi-objective nature of the problem: how to determine the best of the examined candidate configurations. As Fig. 3 shows, we differentiate between hard factors which should be 0 and soft factors that should be also minimized but with a lower priority than the hard factors.

The factors listed in Fig. 3, except for “VM size,” directly relate to costs or constraint violations that need to be minimized. “VM size” has been included because, according to our preliminary experiments, otherwise the algorithm tends to colocate too many components in the same VM. This is logical since – because of the overhead of VMs – mapping the new component to an existing VM is always more energy-efficient than creating a new VM for it. However, having too large VMs may become a disadvantage in the long run, leading to fragmentation of the available PM capacity and hindering the colocation of future components with existing ones even if this were really necessary (because of colocation constraints or license fees). Therefore, since the algorithm makes online decisions based on current objective values without seeing the future, it was necessary to include VM size as a minimization objective to neutralize the energy bias and develop a more future-proof mapping.

For each examined candidate configuration and each optimization objective, we compute the difference that the given selection and/or placement decision would have on the given metric. Based on these atomic metrics, two compound metrics are computed for each examined candidate configuration: the sum of the hard factors and the weighted sum of the soft factors (cf. Fig. 3). For the soft factors, weighting is reasonable because power consumption values, license fees, VM sizes and numbers of violations are of different orders of magnitude, so they should be scaled to the same range to allow a meaningful comparison later on. The weight values should thus be chosen depending on the range of license costs, power consumption values etc. The weights can also be used to express differences in the importance of the individual soft factors. For the hard factors, weighting is not necessary (although possible) because all factors are numbers of violations.

To decide whether candidate configuration \( x \) is better than candidate configuration \( y \), we use the following relation:

\[
x \prec y \Leftrightarrow \text{hard}(x) < \text{hard}(y) \lor \text{soft}(x) < \text{soft}(y),
\]

where \( \text{hard}(\cdot) \) and \( \text{soft}(\cdot) \) denote the two compound metrics defined above.

Putting all pieces together, Algorithm 2 shows the body of the combined algorithm. It should be noted how VM selection and VM placement are interleaved in this algorithm, since each examined configuration encodes both a VM selection and a VM placement decision.

### 4.2 Separate VM selection and VM placement

For comparison, we also develop two policies for VM selection (without VM placement) and two policies for VM placement (without VM selection). Any VM selection policy can be concatenated with any VM placement policy, leading to four different algorithms for deploying a new component.

#### Algorithm 2 The combined algorithm for adding a new component \( c \)

1. \( (V^*, P^*) \leftarrow \text{CANDIDATES}(c) \)
2. \( \textbf{for all } v \in V^* \textbf{ do} \)
3. \( \quad \textbf{for all } p \in P^* \cup \{p(v)\} \textbf{ do} \)
4. \( \quad \text{Compute atomic objectives for } (v, p) \)
5. \( \quad \text{Compute compound objectives for } (v, p) \)
6. \( \textbf{end for} \)
7. \( \textbf{end for} \)
8. \( (v, p) \leftarrow \text{best examined configuration according to } \prec \)
9. \( \text{if } v \text{ is a new VM then} \)
10. \( \quad \text{Start new VM on } p \)
11. \( \text{else if } p(v) \neq p \text{ then} \)
12. \( \quad \text{Migrate } v \text{ from } p(v) \text{ to } p \)
13. \( \textbf{end if} \)
14. \( \text{Deploy } c \text{ on } v \)

#### Algorithm 3 The informed policy for selecting a VM for the new component \( c \)

1. Let \( v^* \) be a new VM with OS compatible with \( c \)
2. \( V^* \leftarrow V \cup \{v^*\} \)
3. \( \textbf{for all } v \in V^* \textbf{ do} \)
4. \( \quad \text{Compute atomic objectives for selecting } v \text{ for } c \)
5. \( \quad \text{Compute compound objectives for selecting } v \text{ for } c \)
6. \( \textbf{end for} \)
7. \( v \leftarrow \text{best examined VM according to } \prec \)
8. \( \text{if } v = v^* \text{ then} \)
9. \( \quad \text{Start new VM} \)
10. \( \textbf{end if} \)
11. \( \text{Return } v \)

#### 4.2.1 Dedicated selection policy

Our first VM selection policy always creates a new, dedicated VM for the new component. Despite its simplicity, this selection policy is quite powerful because it does not create any unnecessary dependence between components, thus leaving full flexibility to the subsequent placement as well as future re-optimizations by live migration. Accordingly, this approach has been used by some previous works [17], [21]. The obvious drawbacks of this policy are the relatively high overhead stemming from the high number of VMs and the lack of colocation for components that must or should be colocated.

#### 4.2.2 Informed selection policy

To remedy the shortcomings of the dedicated selection policy, we devise a much more sophisticated policy aiming to make a well-informed decision on whether to colocate the new component with existing components or to deploy it in a new VM.

As shown in Algorithm 3, the informed VM selection policy closely resembles the combined algorithm. The differences stem directly from the fact that the informed policy does not account for the placement: hence, it investigates only the possible VMs, not pairs of VMs and PMs. Note also that the informed policy examines all the \( |V| + 1 \) possible VMs, whereas the combined algorithm had to sample from its much larger search space to remain fast.

The biggest difference is in the way the objectives are computed. From the metrics shown in Fig. 3, the “Nr. of PM overloads,” “Nr. of violated must / should hardware affinities,” and “Power consumption” objectives are not
applicable at the VM selection stage and are thus ignored in the INFORMED policy. As regards license costs, only VM-based licenses can be considered. VM-level colocation constraints can be fully evaluated, but concerning PM-level colocation constraints, we can only be sure about a violation in case of a must not or should not constraint (if the involved components are mapped to the same VM); for a PM-level must or should constraint, a violation cannot be determined at the VM selection stage. The “VM size” metric can be of course fully evaluated.

Because of the – soft – aim of minimizing VM size, components will be only colocated if this is necessary or advantageous for satisfying colocation constraints or for minimizing license fees.

4.2.3  **Black-Box Placement Policy**

The placement policy receives as input the VM returned by the preceding VM selection policy, which may be a new or an existing VM. The placement policy determines a PM for this VM. In case of an existing VM, this means that the placement policy may decide to migrate the selected VM. This is in line with the combined algorithm, which can also migrate the VM selected for the new component.

The Black-Box placement policy does not consider the components within the VM to place, only its size. This is the same approach as taken by most previous works in the area of VM placement. As suggested by several researchers (e.g., Beloglazov and Buyya [5]), we use the best-fit heuristic to choose the PM that has enough capacity to host the VM but with the minimum remaining free capacity. The VM is then placed on this PM.

Recall that the capacity of a PM is a multi-dimensional vector. For comparing the free capacity of two PMs, we first convert them to single numbers. For this purpose, we take the minimum of the coordinates of the vector. In our previous work we also compared some other metrics for this purpose and found that the minimum metric gives good results [33].

Since this placement policy only considers the size of the VM, we can expect that it will lead to a good placement in terms of energy consumption and number of overloads, but will perform poorly in terms of license costs and conformance with colocation and hardware affinity constraints.

4.2.4 **White-Box Placement Policy**

To address the shortcomings of the Black-Box placement policy, we devise a more sophisticated placement policy that also considers the relations of the components within the VM to be placed. Similarly to the INFORMED selection policy, the idea is again to mimic the combined algorithm as much as possible, now at the level of VM placement.

As shown in Algorithm 4, this involves examining all PMs as possible hosts for the VM and choosing the best one in terms of the investigated objectives. From the objectives of Fig. 3, now all atomic metrics are relevant except for “VM size.” In terms of license costs, only PM-based licenses are relevant at this stage; similarly, from the colocation constraints, only PM-level constraints are relevant. The other metrics are fully evaluated.

It should be noted that the INFORMED selection policy and the White-Box placement policy together base their decisions on the same set of information as the combined algorithm and also the way they examine and compare possible candidates is analogous. However, there are two main differences. First, the combined algorithm examines VM-PM pairs, i.e., it considers selection and placement together, whereas in the catenation of INFORMED and White-Box, first only VMs are considered until one VM is selected, and then only PMs are considered for the already selected VM. This can be seen as an advantage of the combined algorithm. Second, both the INFORMED selection policy and the White-Box placement policy consider all their possible choices (all VMs respectively all PMs), whereas the combined algorithm only examines a subset of the possible candidate configurations, so that it remains fast. The more thorough search can be seen as an advantage of the INFORMED and White-Box policies.

**Algorithm 4** The White-Box policy for placing a VM $v$

1: for all $p \in P$ do
2: Compute atomic objectives for placing $v$ on $p$
3: Compute compound objectives for placing $v$ on $p$
4: end for
5: $p \leftarrow$ best examined PM according to $\prec$
6: if $v$ is a new VM then
7: Start new VM on $p$
8: else if $p \neq p(v)$ then
9: Migrate $v$ from $p(v)$ to $p$
10: end if

5 Evaluation

Our aim is to compare the different approaches to VM selection and VM placement:

- Decoupled VM selection and VM placement, as in most existing approaches (Dedicated+Black-Box)
- Partial integration:
  - VM selection also considers VM placement but not vice versa (INFORMED+Black-Box)
  - VM placement also considers VM selection but not vice versa (Dedicated+White-Box)
- Semi-integrated: VM selection considers VM placement and vice versa (INFORMED+White-Box)
- Fully integrated VM selection and VM placement (Combined)

5.1 Setup

Algorithms for VM placement and VM selection are usually evaluated either with a real cloud or by means of simulation. Using a real cloud is of course more realistic but it comes with several limitations. In particular, it is difficult to experiment with many different parameter settings or to scale the size of the experiment if a real cloud is used. Simulations are much more flexible and hence more popular for research on cloud resource management [38], [41], [48], [49]. Since we would like to compare several different algorithms under many different settings, a simulation-based approach is more appropriate. To still obtain practically relevant results, we used real-world test data, leading to a good compromise between a real cloud and pure simulation.
We have implemented all algorithms presented in Section 4 in a C++ program. To foster reproducibility, this program is freely available from https://sourceforge.net/p/vm-alloc/crosslayer.

In addition to the selection and placement algorithms discussed so far, the program also features a re-optimization algorithm which is invoked regularly and uses VM live migrations to adapt the placement to workload changes. The re-optimization algorithm works as follows: it takes a random VM and uses the WHITE-BOX placement policy to optimize its placement. This optimization step is repeated \( k_r \) times, where \( k_r \) is a given constant.

For component sizes, we used a real workload trace from the Grid Workloads Archive, namely the AuverGrid trace, available from http://gwa.ewi.tudelft.nl/datasets/gwa-t-4-auvergrid. From the trace, we used the first 10,000 tasks that had valid CPU and memory usage data. The simulated time (i.e., the time between the start of the first task and the end of the last one) is roughly one month, thus giving sufficient exposure to practical workload patterns.

As PMs, we simulated HP ProLiant DL380 G7 servers with Intel Xeon E5640 quad-core CPU and 16 GB RAM. Their power consumption varies from 280W (zero load) to 540W (full load) [23]. Throughout the experiments, we focus on two resource types: CPU and memory, i.e., \( d = 2 \). Concerning virtualization overhead, previous work reported 5-15% for the CPU [50] and 107-566 MB for memory [13]. In our experiments, we use 10% CPU overhead and 300 MB memory overhead. The VM placement is re-optimized every 5 minutes. Similarly, constraint violations are also checked every 5 minutes.

Each reported result is the average of 10 runs.

### 5.2 Component sizes only

In our first experiment, components are only characterized by their sizes, i.e., there are no license fees, colocation constraints, nor hardware affinities, and each component has the same OS. This is similar to the evaluation setup of most previous works.

The results – according to the relevant metrics – are shown in Table 2. As can be seen, all algorithms result in 0 overloads. In terms of energy consumption, the COMBINED algorithm has a clear advantage over the others; the results of the others are very close to each other. In particular, the used selection policy has practically no effect. This is indeed true because in this case the INFORMED policy has no reason to colocate multiple components in the same VM, hence it also starts a dedicated VM for each component. The WHITE-BOX placement policy performs slightly better than the BLACK-BOX policy. Since the components are characterized only by their sizes, there is not much difference between the two placement policies. The difference is only that BLACK-BOX uses the best-fit heuristic whereas WHITE-BOX chooses the PM based on its real power consumption.

The advantage of the COMBINED algorithm over the second best in terms of energy consumption is 219.31 kWh, or roughly 2.6%. The average electricity price in the Euro area for industrial customers amounted to 0.125 Euro per kWh in 2015. Thus, the savings translate to 27.1 Euro. Scaling it to a data center with 10 thousand PMs, considering a 12-month period, and assuming a PUE (power usage effectiveness) of 1.7, which is a typical value, the total savings would amount to over 240,000 Euro per year.

In terms of the number of migrations (fourth column of Table 2), there is a clear difference between the algorithms. This could be important because too many migrations could lead to performance degradation or could even make the system unstable [16], [42]. However, relative to the length of the simulation, the number of migrations is actually quite low for all algorithms; even for the COMBINED algorithm which leads to the highest number of migrations, the average number of migrations per PM per day is only 3.32, which should not cause any problems.

Similarly, the COMBINED algorithm takes considerably more time (see last column of Table 2) than the other methods, but with an average execution time of 4.0 milliseconds, it can be still considered fast enough.

### 5.3 License fees

In the next set of experiments, we investigate the effect of license fees. For this purpose, the components are enriched with randomly generated license information.

First, we assume 10 different PM-based licenses (and no VM-based licenses). For each of them, the license fee is randomly chosen between 100 and 1000. We varied the number of components having a license from 2% to 10% of all components; each of the license-relevant components is associated to one of the 10 licenses, taken randomly. The resulting total license fees achieved by the different algorithms are depicted in Fig. 4(a). (Other metrics are not shown because they are very similar to the values from Table 2; in particular, all algorithms lead to 0 overloads.)

The figure clearly shows the superiority of the COMBINED algorithm over all others. The difference keeps growing with increasing number of license-relevant components; if 10% of all components have a PM-based license, then the COMBINED algorithm achieves 44% lower license fees than the best result of the other algorithms. Among the other algorithms, there is no clear winner.

Fig. 4(b) shows the results of the same experiment but with 50 instead of 10 PM-based licenses. Again, the COMBINED algorithm leads to the best results in most cases and its advantage grows with increasing number of license-
The next set of experiments evaluates the impact of colocation constraints. In each experiment, 100 colocation constraints were generated for randomly selected components.
Table 3 shows the results in a condensed form. Each column corresponds to one experiment. For example, in the experiment of the second column, all colocation constraints were PM-level and of type must; in the 8th column, all colocation constraints were VM-level and of type should not. While in most columns, all colocation constraints were on the same level and of the same type, the last column is different: it is a mix of the 8 combinations of colocation level and type, where each combination is present with approximately the same number of constraints. For each experiment, we report the sum of the number of colocation constraint violations and the number of overloads.

For PM-level must and should colocation constraints (second and third column of the table), the COMBINED algorithm is clearly superior to all others, and all other algorithms achieve similarly poor results. Looking more precisely into the operation of the algorithms, the following can be understood about the reasons:

- The INFORMED selection policy has no incentive to colocate multiple components in the same VM since in these experiments the colocation constraints are all on the PM level. As a result, it creates a dedicated VM for each component. This is why there is no significant difference between the results of the two VM selection policies.

- Since both VM selection policies create small VMs, this leads to low fragmentation. Therefore, when a PM-level colocation constraint motivates the WHITE-BOX placement algorithm to place the new VM on the PM where one of the already placed components resides, it will often not succeed because the given PM does not have sufficient free capacity. This is why there is no significant difference between the results of the two VM placement policies.

- The COMBINED algorithm on the other hand, when confronted with this situation, will put the new component into the same VM as its peer and then migrate the VM containing both components to a PM with sufficient free capacity. Note that the other approaches also have this option, but do not choose it because of the separate evaluation of the selection and placement possibilities.

Concerning the PM-level should not and must not constraints, the results are more easily understood. The COMBINED algorithm as well as the WHITE-BOX placement policy are able to avoid constraint violations by not placing the new component (respectively the VM where it has been put) onto the same PM as some other component(s). The BLACK-BOX placement policy, which does not consider colocation constraints, necessarily leads to some violations. It is interesting to note that the number of violations is now much lower than in the case of must and should constraints. This is not surprising though: placing the new VM on a random PM has a high chance to meet a should not or must not constraint if the number of “bad” PMs is low, whereas meeting a must or should constraint has much lower probability.

For VM-level must and should colocation constraints, the DEDICATED VM selection policy obviously leads to poor results since it never selects the same VM for components that should be colocated. In fact, these results are even significantly worse than in the case of the similar PM-level constraints, since in this case definitely all constraints will be violated, whereas in the case of PM-level constraints, PM-level colocation was still possible. The INFORMED selection policy, on the other hand, puts all components that must or should be in the same VM indeed into the same VM. Together with the WHITE-BOX placement policy, this leads to very good results, similar to those of the COMBINED algorithm. For VM-level must colocation constraints, it even improves on the results of the COMBINED algorithm.

For VM-level should not and must not colocation constraints, as can be seen, all tested algorithms achieve optimal results. Not colocating some components in the same VM is very easy, for example by using dedicated VMs for each component.

Over all experiments with colocation constraints, the COMBINED algorithm gives excellent results: with the exception of the VM-level must constraints, where it ranks only second after the INFORMED+WHITE-BOX combination, it always gives the best results, in several cases dramatically better results than any other algorithm. This is also mirrored in the last column of Table 3, showing the combined effect of different colocation constraints. Here, too, the COMBINED algorithm is the clear winner, leading to more than an order of magnitude better results than all other algorithms. From the results of the remaining algorithms it is also apparent that the INFORMED VM selection policy has a clear advantage over the DEDICATED policy. This is not surprising, given the inability of the DEDICATED policy to appropriately handle VM-level must or should colocation constraints.

### 5.5 Hardware affinity

In this set of experiments, the PMs were enriched with PM feature information and the components were enriched with hardware affinity requirements. In particular, we define $k_f$ PM features and each PM has each feature with probability $p_f$. Each component requires (must relationship) each PM
TABLE 4
Number of hard hardware affinity constraint violations plus number of overloads, depending on the number of PM features

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Nr. of PM features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>COMBINED</td>
<td>0</td>
</tr>
<tr>
<td>DEDICATED+BLACKBOX</td>
<td>11,745</td>
</tr>
<tr>
<td>DEDICATED+WHITEBOX</td>
<td>0</td>
</tr>
<tr>
<td>INFORMED+BLACKBOX</td>
<td>11,973</td>
</tr>
<tr>
<td>INFORMED+WHITEBOX</td>
<td>0</td>
</tr>
</tbody>
</table>

In the next experiment, we considered a single PM feature and varied the percentage of PMs offering this feature from 3% to 21%. In each run, 1% of all components required the given feature while 5% of all components were defined with hardware affinity requirements. On the other hand, the B-PLACE policy does not achieve excellent results, irrespective of the used VM selection policy. The COMBINED algorithm achieves similarly good results. It should also be noted that increasing \( k_f \) leads to a lower number of hardware affinity requirements, for the B-PLACE placement policy, as shown in Fig. 5.

As can be seen, if only few PMs possess the required feature, then all algorithms lead to a high number of violations; it is probably not even theoretically possible to fulfill all hardware affinity requirements. As the number of PMs with the given feature increases, the results form two clusters: the COMBINED algorithm and the algorithms using the WHITEBOX placement policy manage to use these PMs to fulfill a growing number of hardware affinity requirements, leading to a steady decrease in the number of hard constraint violations. On the other hand, the BLACK-BOX policy hardly benefits from the increased availability of “good” PMs, in line with the fact that this VM placement policy does not explicitly consider hardware affinities. Within the two clusters, there are no significant differences, suggesting that VM selection does not considerably influence the satisfaction of hardware affinity requirements.

A very similar pattern can be observed also in Fig. 6. In this experiment, a single PM feature is considered which is offered by 20% of all PMs. The fraction of components requiring the given feature is varied from 1% to 10%. As can be seen in the figure, the BLACK-BOX VM placement policy again leads to a high number of constraint violations, which steeply increases with the growing number of components with hardware affinity requirements. On the other hand, the WHITE-BOX placement algorithm and the COMBINED algorithm can solve the problem with a much lower number of constraint violations.

### 5.6 Number of operating systems

We also varied the number of operating systems and let each component require a randomly selected OS. However, the number of the considered operating systems did not have a noticeable effect on the used quality metrics.

### 5.7 Putting the pieces together

So far, the effect of different aspects was investigated in isolation. In practice, all aspects may be present at the same time. Therefore, we also present the results of an experiment in which multiple aspects are applied as follows:

- **License fees:**
  - Number of PM-based licenses: 10
  - Number of VM-based licenses: 10
  - Ratio of components with a license: 5%

- **Colocation constraints:**
5.8 Effect of search space limitation

As explained in Section 4.1, the COMBINED algorithm considers only a subset of all possible PM-VM pairs, so that it remains fast. Now we investigate the effect of this limitation by comparing it against a version of the algorithm, called COMBINED-FULL, that considers all possible PM-VM pairs.

We have repeated the experiment of Section 5.7 also with the COMBINED-FULL algorithm. As Table 5 shows, COMBINED-FULL achieves an improvement of 5.5% in the number of violations of hard constraints over COMBINED. However, this comes at the expense of a comparable increase in license costs as well as a more than 7 times increase in the algorithm’s execution time. This shows that the method used in COMBINED to determine sensible candidate solutions represents a good trade-off between execution time and solution quality.

6 Conclusions

Based on the measurement results of Section 5, we can now try to answer the original questions regarding the benefits of (i) doing VM selection and VM placement together and (ii) including information about VM placement in VM selection and about VM selection in VM placement. For this purpose, we first summarize the empirical results:

- For PM-based licenses, the COMBINED algorithm results in up to 44% lower license fees than the second best algorithm.
- For VM-based licenses, the COMBINED algorithm and the INFORMED VM selection policy lead to significantly lower license fees than the DEDICATED policy. However, in the case of the INFORMED policy, this comes at the cost of PM overloads.

6. It should be noted that also COMBINED-FULL is a heuristic. It considers all possible solutions that use a single migration, but it does not consider performing multiple migrations.
Altogether, we can conclude that the COMBINED algorithm is in all cases among the best-performing algorithms. Sometimes it clearly outperforms all other methods (e.g., for PM-based licenses), in other cases it is just one of the best performers. However, in none of the test cases was it clearly inferior to another algorithm. So the answer to the first question is clear: combining VM selection and VM placement in a single optimization algorithm leads to significant benefits compared to the isolated treatment of the two problems as in most existing works (DEDICATED+BLACK-BOX).

Regarding the second question, we can state that the WHITE-BOX VM placement policy clearly outperforms the BLACK-BOX policy. Hence, it is advantageous to include component-level information in VM placement decisions. The relationship between the DEDICATED and INFORMED VM selection policies is less clear because in many cases, the difference between their results was marginal. However, there were some test cases where INFORMED led to clearly better results: for VM-level must and should colocation constraints and, even more importantly, when all types of constraints were applied at once. Thus we conclude that, for these types of scenarios, it is also advantageous to include in VM selection decisions foresight into VM placement.

In terms of the current state of the art, it should be underlined that existing approaches almost exclusively focus either on VM selection or VM placement, ignoring the other problem. Hence, the DEDICATED+BLACK-BOX algorithm embodies the current state of the art. Compared to this, our results show that the tighter the two problems are integrated, the better results can be achieved.

The comparison between the fully integrated approach (COMBINED) and the semi-integrated approach (INFORMED+WHITE-BOX) is tricky because none of them dominates the other in all considered metrics. COMBINED was better in handling licenses and PM-level colocation constraints, the two approaches yielded similar results for handling VM-level colocation constraints and HW affinity constraints, while INFORMED+WHITE-BOX led to fewer migrations and had lower execution time. The number of migrations and the execution time of COMBINED are also in an acceptable range, and so, since it leads to lower license costs and fewer constraint violations, we consider it preferable. Also in the experiment which contained all the investigated aspects, the COMBINED approach led to better results according to all measured metrics.

A further question is whether there is any “hidden” overhead of the proposed approach, since it leads to more co-locations and as VM isolation is not perfect, this could lead to performance degradation. Fortunately, our model supports anti-colocation constraints with which co-location of components that would interfere with each other can be prohibited. Therefore, the proposed approach will only co-locate components that do not interfere.

ACKNOWLEDGMENTS
This work was partially supported by the Hungarian Scientific Research Fund (Grant Nr. OTKA 108947) and the European Union’s Horizon 2020 research and innovation programme under grant 731678 (RestAssured).

REFERENCES


Zoltán Ádám Mann received the MSc and PhD degrees in Computer Science from Budapest University of Technology and Economics in 2001 and 2005, respectively. He is currently senior researcher at paluno – The Ruhr Institute for Software Technology, University of Duisburg-Essen. His research interests include optimization problems and algorithms in cloud computing.
A.1 VM sizing

The following aspects of the trilateral component–VM–PM mapping must be taken into account in relation with the sizing of VMs:

- **Placeability.** As demonstrated by the example of the Introduction in the paper, the way components are mapped to VMs influences the size of the resulting VMs, which in turn determines the possible placements of the VMs on the PMs and hence the costs of the placement. It is difficult to make good decisions in VM selection because the placeability of a VM depends not only on its own size, but also on the size of other – existing and future – VMs.

- **Possibilities of live migration.** VMs provide a generic and transparent facility for migration in order to adapt to changes in the workload. Although some applications may offer the possibility to move individual components from one VM to another, this cannot be assumed in general. Hence in this paper we assume that only VMs can be migrated. The VMs are thus the unit of migration, which means that the VMs' sizes determine the granularity of migrations [4]. From this point of view, it is beneficial to have small VMs because they allow a fine-granular control of the PMs' utilization, thus avoiding fragmentation and achieving near-optimal utilization.

- **Overhead.** Virtualization introduces some overhead in terms of resource consumption. Since every VM adds some overhead (e.g., the size of the guest operating system), from this point of view it is beneficial to have a lower number of larger VMs. Obviously, this contradicts the above aspect which would lead to many small VMs, so that a good balance has to be found between the two aspects.

A.2 Co-location

The following aspects all lead to some kind of constraints or preferences on co-location of components in the same VM and/or PM:

- **Communication.** For components that communicate with each other, it may be necessary or at least advantageous to map them to the same VM, or at least to VMs on the same PM [5]. For example, a legacy application (i.e., one that was not developed with a multi-VM deployment in mind) may consist of multiple components that communicate via shared memory; in this case, these components have to be mapped to the same VM. If they communicate for instance through TCP sockets, then such a colocation is not necessary, but for communication-intensive applications it may still be advantageous in order to reduce latency and save network bandwidth.

- **Security.** Although virtualization provides a level of isolation between co-located VMs, it does not provide sufficient defense against malicious attacks. Using covert channels in hardware or software, a malicious VM can gain sensitive information from co-located VMs [2]. Therefore it is important to co-locate VMs hosting critical components only with VMs hosting only trusted components. In a private cloud, security concerns are typically lower than in public clouds, but in critical domains (e.g., banking), it is still important to isolate critical productive components from the ones whose security is not guaranteed.

- **Fault tolerance.** For some components, high availability may be necessary to improve reliability [6]. This has two important consequences for selection and placement. First, such components should not share a VM with other, less stable components to avoid that the failure of another component crashes the VM. (In contrast, they may be on the same PM since a VM can tolerate the crashing of a co-located
Second, if a component is replicated to guarantee high availability, then the replicated instances should be placed on different PMs so that a PM fault impacts only a single instance.

- **Performance interference.** Components in the same VM compete directly for all system resources. Between VMs, the virtualization layer partitions some resources (e.g., memory space), but this isolation is far from perfect, so that for some resources – like memory bandwidth or caches – there is also competition between co-located VMs, which may lead to significant performance degradation [1]. Therefore, components that use the same resource intensively, should be packed into distinct VMs, possibly also on distinct PMs.

- **Correlated load peaks.** Sudden load increases are dangerous in servers with high utilization because resource overloads may easily lead to SLO violations. It is especially problematic if the load of multiple components on the same PM or VM increases at the same time. Therefore, co-location of several correlated components should be avoided [3].

### A.3 Other affinity aspects

The following aspects, leading to software or hardware affinity preferences, must also be taken into account when mapping components to VMs and VMs to PMs:

- **Operating system dependency.** In many cases, application components depend on a specific operating system (OS), a specific version (or range of versions) of an OS, or are compatible only with a set of OSs. While the OS of the VM can be chosen independently from the host OS, the OS of the VM must match the OS requirements of the components.

- **Hardware affinity.** Some components may require some special hardware feature, or there may be a preference for such. For example, a component may run only on a PM with a GPU\(^1\) of a given vendor, or it may be able to take advantage of such hardware to boost its performance.

- **Licensing.** Some components may require costly licenses. There are many licensing models, several of which are agnostic of placement: e.g., if license cost depends on the number of users, then the placement of the component is irrelevant. However, some licensing constructs specify fees depending on the number of machines (either VMs or PMs) on which the software runs, in some cases also weighting the number of machines with some coefficients of computing power. For such licensing models, the placement of the components does matter: colocating multiple components with the same license on the same VM or PM leads to a reduction of the license fees to be paid.

### A.4 A note on component sizing

In this work, we focus on software deployment. At this stage we can assume that the components and their sizes (i.e., resource requirements) are given. The components and their sizes must be determined in an earlier phase of the software development process in such a way that the performance of the application complies with the SLOs that it has to fulfill. This is a challenging task, but it is outside the scope of this paper.

### REFERENCES


---

1. Graphical Processing Unit
Modeling Data Protection Vulnerabilities of Cloud Systems using Risk Patterns

Published in the Proceedings of the 10th System Analysis and Modeling Conference (SAM), pp. 1-19, 2018

Alexander Palm, Zoltán Ádám Mann, and Andreas Metzger

paluno – The Ruhr Institute for Software Technology
University of Duisburg-Essen, Essen, Germany

Abstract. Ensuring the protection of sensitive data is important for the adoption of cloud services. Cloud systems are becoming increasingly complex and dynamic, leading to various potential scenarios for attackers to get access to sensitive data. To handle such data protection risks, the concept of risk patterns was introduced previously. A risk pattern models a structural fragment of cloud systems that should not appear in the running system because it would lead to high data protection risks. At deployment and at run time, graph pattern matching and dynamic re-configuration methods can be used to ensure that the run-time model of the cloud system contains no instance of the risk patterns.

The previous work left it open, however, how and to what extent real data protection vulnerabilities can be modeled in the form of risk patterns. Therefore, this paper focuses on the design of risk patterns based on vulnerabilities described in the literature. Based on an analysis of 87 papers, we determined 45 risk patterns. Our findings (i) demonstrate that risk patterns can indeed capture many of the vulnerabilities described in the cloud literature, (ii) give insight into the typical structure of risk patterns, and (iii) show the limits of the applicability of the risk pattern approach.

Keywords: Cloud computing, Data protection, Privacy, Run-time model, Risk pattern

1 Introduction

Cloud computing is increasingly popular, thanks to the benefits it brings to both providers and users of cloud services. However, outsourcing sensitive data to the cloud puts the data at a risk, which many users of cloud services are not ready to accept [16].

Data protection in the cloud is hard because cloud systems are increasingly complex and dynamic. They consist of many different physical and virtual machines, as well as various applications and their software components, all of which...
interact and may dynamically reconfigure during run time [1, 9, 15, 30]. In addition, a multitude of stakeholders may be involved, such as service consumers, cloud providers, data subjects, data controllers, and actual end users. Due to such complex interactions, a cloud system may expose vulnerabilities that enable attackers to gain access to sensitive data stored in the cloud. Moreover, since the attributes and interactions of the cloud entities continuously change, data protection vulnerabilities may arise during operation. By data protection vulnerability, we mean the possibility of unauthorized access to sensitive data. This is not the same as a system vulnerability (e.g., if a vulnerable system neither stores nor has access to sensitive data, then there is no data protection vulnerability), but system vulnerabilities may lead to data protection vulnerabilities which put sensitive data at risk.

To identify and mitigate data protection risks in complex and dynamic cloud systems, we have introduced the concept of risk patterns in our earlier work [26]. That approach was based on two types of artefacts:

- A model of the – current or planned – configuration of the cloud system, including infrastructure elements, middleware, applications, data, and the involved actors;
- A set of risk patterns, which describe cloud configurations that would cause too high risks of data protection violation and hence must be avoided.

For modeling the configuration of the cloud system, a meta-model was proposed [17]. When a cloud system is to be deployed, the system designer creates the model of the planned configuration as an instance of the meta-model. When the configuration changes during the deployment process or later during the operation of the system, the model is updated accordingly, so that it always reflects the current state of the cloud system and can be used as a run-time model.

Risk patterns are expressed in a domain-specific language based on the same meta-model as the cloud model. Risk patterns model fragments of a cloud system by specifying the presence or absence of certain entities, attributes, or relations. Risk patterns capture forbidden fragments of a cloud system model that would exhibit overly high data protection risks. During deployment and at run time, the model of the cloud system is checked for the existence of fragments corresponding to risk patterns. If an instance of a risk pattern is found in the cloud model, a potential data protection vulnerability is identified, which may be mitigated with appropriate changes of the deployment or by run-time adaptation.

Our previous work [26, 17] evaluated the risk pattern approach using two example risk patterns. The evaluation showed that, if the relevant data protection vulnerabilities are captured in the form of risk patterns, then these risk patterns can indeed be used to detect and mitigate the data protection risks during deployment and at run time. The prerequisite is a catalog of risk patterns capturing the relevant data protection vulnerabilities. Our previous work did not address in detail how risk patterns can be devised, leaving several questions open:

- Is it feasible to model a broad range of real data protection vulnerabilities in the form of risk patterns?
– What is the typical size and structure of risk patterns? (This is important as it impacts the applicability of graph pattern matching algorithms in terms of their computational complexity (which in turn is not part of this paper))
– For which kinds of data protection vulnerabilities is the risk pattern approach appropriate?

This paper seeks to answer these questions by gaining experience with modeling risk patterns. Specifically, we review 87 papers from the cloud security literature (which were collected in a previous survey [3]) and identify the ones that describe relevant vulnerabilities in sufficient detail. Then, we devise risk patterns for the vulnerabilities described in these papers. This results in a total of 45 risk patterns.

Our findings show that most of the vulnerabilities that were described in sufficient detail in the respective papers could indeed be captured by appropriate risk patterns, thus demonstrating the general applicability of the risk pattern approach. All identified risk patterns share the same high-level structure and consist of 6 to 10 entities. This suggests that graph pattern matching can indeed be efficiently used to find risk patterns in cloud models. Also some limitations of the risk pattern approach are uncovered, relating to both the types of vulnerabilities that can be captured (e.g., vulnerabilities resulting from human and social aspects are not appropriate) and the underlying cloud meta-model (a very fine-grained meta-model can lead to a proliferation of many similar risk patterns to capture essentially the same vulnerability).

The remainder of this paper is organized as follows. In Section 2 we review the meta-model underlying the risk pattern approach. Section 3 then gives an overview of the methodology used to define the risk patterns and Section 4 presents the structure of risk patterns. In Section 5 we describe the risk patterns that we derived from the literature. Section 6 summarizes the lessons learned during the process, while Section 7 describes related work and Section 8 concludes the paper.

2 Cloud Meta-Model

In this section we briefly review the previously proposed meta-model [17]. The model of the cloud system, which plays a central role in the risk pattern approach, is an instance of this meta-model. Further, the risk patterns also reference entities, attributes, and relations from this meta-model.

The meta-model consists of the packages Actors and Assets (see Fig. 1). The Actors package defines the different data-specific roles (e.g., data subject, data controller) and cloud-specific roles (e.g., infrastructure provider) that a natural or legal person can have, and a trust relationship that can exist between different actors. An actor can also access and/or own assets, e.g. an actor can access a virtual machine.

The Assets package is further divided into the sub-packages Data, Applications, Middleware and Infrastructure. The elements necessary to model the data that has to be protected are given by the data sub-package. The main element
Fig. 1. Abstract view of the meta-model for cloud models [17]

Fig. 2. Infrastructure sub-model of the meta-model for cloud models [17]

within this sub-package is the data object. Data can be stored in form of a stored data set or exchanged between different application components via a data flow element. The application sub-package comprises the elements needed to model software elements, like applications with different application components and connectors between them. Middleware elements, like web servers, application servers and database management systems are available in the middleware sub-package. The elements needed to model the infrastructure of a cloud system, like virtual machines (VMs), physical machines (PMs) and data centers (DCs) are given by the infrastructure sub-package. As an example, Fig. 2 shows the contents of the infrastructure sub-package. The full meta-model is shown in the Appendix.

3 Methodology

In this section we describe the methodology that we used to derive a catalog of risk patterns.
The starting point was a survey about cloud security [3]. From the categories defined in that survey paper, we focused on the categories ‘vulnerabilities, threats and attacks’ and ‘cloud security’, and analyzed all papers in those categories which mentioned the aspects confidentiality or privacy. In the publications of the first category, different attacks on cloud systems are described. Nearly all papers of this category (21 out of 23) touch the aspects confidentiality or privacy. Publications of the second category focus on security solutions and do not mention confidentiality or privacy to the extent the publications of the first category do (66 of 81 papers were considered relevant).

In the next step, we removed the secondary literature from the first category, leaving 9 out of 21 papers of this category. We analyzed all remaining papers for their level of detail concerning the description of an attack on or a vulnerability of a cloud system and the underlying cloud architecture. Only two papers of the category ‘cloud security’ described the underlying problem of the security solution in sufficient detail, so 64 papers were removed. In the category ‘vulnerabilities, threats and attacks’, 8 out of 9 papers were detailed enough.
In the end, 10 papers remained that served as a basis for the modeling of risk patterns. An overview of these publications is given in Table 1 and an overview of the literature analysis is given in Fig. 3.

After the analysis of the literature, we derived risk patterns based on the relevant excerpts of cloud architectures described in the selected publications. The modeling of a risk pattern was done in three steps:

1. Analysis and description of the attack
2. Identification of the underlying system vulnerability
3. Identification of the relevant paths within the meta-model

After the analysis and description of the attack, the main goal was to identify the specific system vulnerability exploited by the attack. This includes the identification of elements of the meta-model suitable to model this vulnerability and in particular its attack point. After this, the sensitive data that should be protected are modeled and in the third step the relevant paths connecting the sensitive data with the attack point are identified.

### 4 Structure of Risk Patterns

A risk pattern is a sub-structure of a cloud system configuration, which threatens the protection of sensitive data and therefore has to be avoided [26]. A risk pattern can typically be divided into three parts (see Fig. 4):

- Part (a) represents the personal data that need to be protected. These data are always modeled by the same elements of the data package: a data record which is part of a stored data set, and an actor who is the data subject that the data belong to.
- Part (b) represents the attack point of the system vulnerability: the point of the configuration through which an attacker gets access to the system. This part of the risk pattern depends on the type of the modeled attack.

<table>
<thead>
<tr>
<th>Publication</th>
<th>Category</th>
<th>No. of risk patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Somorovsky et al. [29]</td>
<td>Control Interfaces</td>
<td>12</td>
</tr>
<tr>
<td>Aviram et al. [4]</td>
<td>Side Channel</td>
<td></td>
</tr>
<tr>
<td>Green [12]</td>
<td>Side Channel</td>
<td></td>
</tr>
<tr>
<td>Okamura &amp; Oyama [19]</td>
<td>Side Channel</td>
<td></td>
</tr>
<tr>
<td>Ristenpart et al. [22]</td>
<td>Side Channel</td>
<td></td>
</tr>
<tr>
<td>Zhang et al. [32]</td>
<td>Side Channel</td>
<td></td>
</tr>
<tr>
<td>Rocha &amp; Correia [23]</td>
<td>Privilege Exploitation</td>
<td>7</td>
</tr>
<tr>
<td>Sedayao et al. [27]</td>
<td>Privilege Exploitation</td>
<td></td>
</tr>
<tr>
<td>Bermsmed et al. [5]</td>
<td>Service Mistrust</td>
<td>2</td>
</tr>
</tbody>
</table>
Fig. 4. An example risk pattern, structured into three parts

- Part (c) contains two connections between part (a) and part (b). One connection is between the sensitive data and the attack point, the other one between the actors. The possibilities for the first connection are determined by the underlying meta-model and depend on the ‘distance’ (within the meta-model) between parts (a) and (b). The second connection is always a mistrust relation between the data subject and the attacker. (Note: A dashed line implies that the according relation must not exist, whereas a solid line implies that the according relation must exist)

The attributes of the entities also play an important role. Sometimes vulnerabilities only differ in some attributes.

5 The Devised Risk Patterns

In this section we present how we modeled different types of vulnerabilities from the literature by different categories of risk patterns. The categorization is based on the different attack points of the risk patterns. Overall the risk pattern catalog includes 45 risk patterns in four categories. For reasons of space we include here only some examples. The full risk pattern catalog is available under https://zenodo.org/record/1324125#.W2A2mrhCREY.
5.1 Category: Control Interfaces

The first category of our catalog comprises risk patterns modeling attacks on control interfaces. Control interfaces are interfaces which give users the opportunity of maintaining their resources. The maintenance of resources includes the instantiation, starting and shut-down of virtual machines. Although such interfaces are protected with measures like authorization and signatures, still vulnerabilities exist. To provide the underlying basics for the definition of the risk patterns of this category, we first introduce an attack scenario targeting a vulnerability of a control interface, before we then describe the risk patterns derived from this scenario.

Underlying attacks. The attack scenarios which serve as a baseline for the definition of the risk patterns of this category are described in [29]. To provide the possibility of maintaining resources, Amazon Web Services (AWS) provides mainly two interfaces: a SOAP interface and a Web interface.

The SOAP interface is based on the Simple Object Access Protocol (SOAP) which uses an X.509 certificate for the identification of the user and an XML-based signature to enable authentication and prove the integrity of a message. Furthermore the SOAP messages themselves are based on XML. The authors of [29] proved that SOAP is vulnerable to so-called signature wrapping attacks. To perform a signature wrapping attack on SOAP, the attacker has to intercept a SOAP message exchanged between the user and the interface. Then the attacker can add an additional message body to the intercepted message and reuse the signature. This enables the attacker to perform arbitrary operations on the SOAP interface, because only the body referenced in the signature is verified for integrity, but the additional body is interpreted.

The Web interface enables an attacker to perform a so-called script injection attack on the cloud interface. As this attack is also founded in the underlying protocol (HTTP), the derived risk patterns are analogous to those of the aforementioned attack. Based on these attack scenarios, the usage of SOAP and HTTP can be considered as a data protection vulnerability of a cloud system and therefore is modeled as a risk pattern.

Risk pattern definition. The attack point of the risk patterns of this category is shown in Fig. 5 (Note: this is an excerpt of the corresponding risk pattern shown in Fig. 4). The interface is modeled as a ‘Public IaaS Interface’ entity from the Infrastructure package of the meta-model. The IaaS user accessing the interface may behave as the attacker of the attack scenario described above, thus gaining unauthorized access to sensitive data. The protocol of the interface can be identified by the attribute ‘protocol’, which is ‘SOAP’ in Fig. 5.

The meta-model allows multiple possibilities for connecting the ‘Public IaaS Interface’ of Fig. 5 with the sensitive ‘Data Record’, i.e., multiple possibilities for the attacker to actually access sensitive data, depending on the specific cloud configuration. The risk pattern shown previously in Fig. 4 is one possibility,
in which the access takes place through a chain of a virtual machine (VM), an application component, and a database management system (DBMS). The meta-model allows five further possibilities, described by the following chains of assets:

- VM → application component → local database
- VM → application server → application component → local database
- VM → DBMS
- VM → application server → DBMS
- VM → application server → application component → DBMS

These risk patterns capture different cloud configurations that exhibit a similar data protection vulnerability.

5.2 Category: Side Channels

The second category of our catalog includes risk patterns modeling side channel attacks. Side channels are based on a shared resource (e.g., CPU cache) which enables data leakage or is misused for communication between two virtual machines that are co-located on the same physical machine but belong to different users.

Underlying attacks. Side channels can have two consequences: they can be misused for communication between otherwise isolated VMs [19, 22] or for data leak [12, 32, 11, 22]. Side channels always rely on multi-tenancy, which means that a physical machine is shared among different users.

How such co-location can be accomplished in Amazon EC2 is described in [22]. An attacker can instantiate lots of VMs of the same instance type and inside the same availability zone as the victim’s VM. Doing this, there is a high probability that one of the instantiated VMs is on the same physical machine as the victim’s one. The probability can even be increased if the instantiation process is launched right after the victim’s instance is re-instantiated, because following Amazon EC2’s VM placement strategy, physical machines with free capacity are filled first before new physical machines are started. After co-location is achieved, the way is cleared for one of the following attacks or techniques.
In [19] a technique called CCCV for the use of the CPU load as a side channel is described. The technique is possible if virtual CPUs (VCPUs) of different VMs share the same physical CPU. Assuming a spyware was injected into the victim’s VM, the CPU load can be manipulated, so that data can be transferred adhering to the protocol described in [19]. This protocol is based on the fact that a high CPU load of one VM affects the performance of co-located VMs, and can thus be observed by them. A different attack where CPU load is misused as a side channel for communication is also described in [22].

The Prime+Trigger+Probe (PTP) technique uses a shared cache as side channel [22, 11]. As its name implies, PTP comprises three phases. Within the ‘prime’ phase, the attacker fills all lines of the shared cache and measures the time needed to read each of these lines. In the following ‘trigger’ phase, the attacker hands over the control of the shared cache to the victim’s VM. The victim’s VM then may change some lines of the cache and hand over control back to the attacker’s VM. The change of cache lines results in cache misses when the attacker probes the cache during the ‘probe’ phase. This technique can be used to communicate between the two VMs [22] when a change of the cache is interpreted as the sending of a ‘1’ and no change of the cache is interpreted as a ‘0’. Furthermore, this technique can be used to extract sensitive data such as private keys [12], because the attacker can possibly determine which operations were carried out by the victim’s VM based on the changes of the access times of different cache lines and also based on which cache lines have been changed. A slightly different technique is described in [32] and a scenario stating possible consequences of such an attack is described in [4].

**Risk pattern definition.** As side-channel attacks always rely on a shared resource used by two co-located VMs and only differ in the type of resource used and in nuances of hardware settings, we introduced abstract risk patterns which can be made concrete through attributes of the concerned elements depending on the specific vulnerability that should be exploited. The attack point of the abstract risk patterns (and therefore also of all concrete risk patterns of this category) comprises two VMs being hosted on the same physical machine (see Fig. 6). The actor accessing one of the VMs (and thus serving as attacker) is not trusted by the data subject. Attributes used to concretize abstract risk
patterns include ‘hypervisor’ to specify a certain hypervisor and ‘cpu-scheduling’ to specify a certain kind of CPU scheduler.

5.3 Category: Privilege Exploitation

The third category of our catalog comprises risk patterns modeling privilege exploitation attacks. In privilege exploitation attacks, an administrator abuses their privileges to get access to sensitive data [23, 27].

Underlying attacks. Administrators of cloud systems normally have no rights to log on to client VMs. However, an administrator with root privileges can generate memory dumps of client VMs for troubleshooting. The administrator can also misuse this opportunity to extract private data (e.g., cryptographic keys) from such memory dumps [23]. Although data might be stored encrypted on permanent storage, the administrator can get access to the cleartext if the memory dump is generated at the right moment (i.e. when data are decrypted for processing). Because private keys are often stored as ASN.1-objects, an attacker just needs to search for typical byte sequences of ASN.1-objects within the memory dump. This attack becomes more difficult if secure hardware is used that prevents the memory from being dumped. In this case, an attacker may trigger a VM relocation first, and perform the attack when the VM is relocated on a physical machine which is not using this kind of hardware. A similar kind of attack is also possible on storage devices [27].

Risk pattern definition. The attack point of the risk patterns of this category is shown in Fig. 7. As all attacks of this category are based on an administrator abusing their privileges to access sensitive data, this situation is modeled in the risk patterns of this category. More specifically, an untrusted IaaS operator managing the physical machine on which the VM with the personal data of the data subject is hosted is accessing these data.

5.4 Category: Service Mistrust

The fourth category of our catalog comprises risk patterns modeling the problem of mistrust within service compositions. Because service compositions consist of
Fig. 8. Excerpt of a risk pattern of the category ‘service mistrust’

different services which are composed in a hierarchical fashion and belong to different providers, a data subject may not trust – and may not even know – some of the involved providers. An untrusted provider getting access to sensitive data constitutes a data protection risk.

Underlying attacks. Bernsmed et al. [5] consider a scenario where a service provider might access personal data processed by its service. In service compositions, the situation of trust becomes more complex. Although there may be a chain-of-trust between the participating providers of a service composition, a data subject whose data is processed by the composed service, does not necessarily trust all of the involved providers. The situation becomes even worse if the data subject does not even have knowledge about the participating providers. Therefore this situation is a data protection vulnerability and can be modeled as a risk pattern.

Risk pattern definition. The attack point of the risk patterns of this category is shown in Fig. 8. The situation is modeled by a SaaS (Software as a Service) operator that is responsible for the operation of an application component – which represents a service in a possibly larger composition of services – and therefore can access sensitive data processed by this application component.

6 Lessons Learned

Through the modeling of real data protection vulnerabilities in the form of risk patterns, we have gained insight into both the applicability of the risk pattern approach and the characteristics of typical risk patterns.

6.1 Applicability of the Risk Pattern Approach

Since we managed to represent several real data protection vulnerabilities in a natural way in the form of risk patterns, we can state that the risk pattern approach is appropriate for modeling data protection vulnerabilities. In particular,
quite complex attack scenarios spanning multiple cloud layers could be modeled, and also very different kinds of attacks.

That said, it is important to note that the attacks described in several papers could not be reasonably modeled in the form of risk patterns. In some cases, this was due to a lack of detail about the vulnerabilities in the respective papers (cf. Fig. 3), because those papers focused primarily on describing data protection techniques against some classes of attacks, rather than describing specific vulnerabilities in detail. However, there were also cases where the non-applicability of the risk pattern approach had other reasons, and these reasons shed some light on the limits of the applicability of the approach:

- Some vulnerabilities are not technical, but stem from human or organizational factors, e.g., lack of security training for personnel (e.g. the vulnerability described in [7] lies is the careless behaviour of users not cleaning their Amazon Machine Images of passwords before making them available for others). In contrast, risk patterns are appropriate for capturing forbidden socio-technical configurations that are at least partly technical in the way they expose data.

- Several papers focus on attacks and not on vulnerabilities, e.g., describing several ways an attacker could exploit some basic vulnerability. Risk patterns are not meant to capture specific attacks, but rather configurations that lead to high risks of successful attacks (therefore the risk patterns stemming from different side-channel attacks (cf. Section 5.2) only slightly differ.). Thus, the use of risk patterns is more proactive than, for instance, intrusion detection techniques [28].

- Several papers focus on system vulnerabilities which do not necessarily imply data breach (e.g. attacks compromising the availability of resources, cf. [6, 13]). Although risk patterns could in principle also be used to model such system vulnerabilities, our focus was on data protection vulnerabilities, thus rendering some attacks irrelevant.

- Some papers describe vulnerabilities that are not cloud-related. While risk patterns could in principle also be used in other contexts, the currently used meta-model is cloud-specific, hence we only considered cloud-related vulnerabilities. However, we decided to differentiate between cloud-related side-channel attacks and the more general virtual machine escape (cf. [21]). The latter was not cloud-related for us and therefore excluded.

- Some papers describe unlikely attacks, i.e., attacks that work only under strong assumptions about the possibilities of the attacker. Risk patterns are supposed to be created in the course of risk assessment, covering the configurations that are considered to be too risky – not necessarily all configurations that might allow some attack under unlikely conditions. This also applies to some of the configurations we modeled in this paper.

The applicability of the approach is also strongly related to how cloud configurations can be modeled by using the types from the meta-model described in [17]. The experience with using the meta-model has shown that it is indeed
a solid basis for modeling complex cloud configurations. In particular, the possibility of the meta-model to combine different hardware and software entities, data, and actors in a single model has proven invaluable, since all risk patterns span several of these categories. Some small extensions to the meta-model were also necessary, e.g., some new attributes and relationships between certain assets and actors had to be introduced (e.g. to model the direct access of an IaaS user to a virtual machine or a DBMS running directly on a virtual machine). This is normal; project-specific tailoring of the meta-model was also envisaged in [17].

6.2 Characteristics of Risk Patterns

As shown in Section 4, risk patterns have a common structure. All risk patterns that we devised follow this same structure. From a graph-theoretic point of view, a risk pattern defines a path from an attacker to the sensitive data (the path through which the attacker may be able to access the data), plus an additional path between the same two vertices encoding that the data belong to a data subject who does not trust the attacker. This means that instead of a general graph pattern matching problem as suggested in [26], only subgraphs of very limited structure (cycle graphs) must be searched for in the cloud model, which may require significantly less computation.

For assessing the computational implications, it is also an important finding that the risk patterns are quite small: all the devised risk patterns consist of 6 to 10 entities, with exactly two connections per entity.

Another aspect is the number of risk patterns. In particular, a single system vulnerability leads to multiple risk patterns: if an asset is compromised from which there are $k$ different kinds of paths to the sensitive data, then potentially $k$ risk patterns are needed to capture all possible data protection vulnerabilities stemming from the same system vulnerability (and it is possible to use different data protection mechanisms to protect each of those paths). It is important to note that the number $k$ of different kinds of paths from an asset to the data depends on the meta-model and especially the possible connections between the different entities of it. With the meta-model used in this work, $k \leq 6$. Moreover, for some system vulnerabilities, indeed 6 different risk patterns were needed; for some other system vulnerabilities, a lower number of risk patterns was sufficient. If the meta-model were refined with further types and relationships, this could lead to higher values of $k$ and thus to a proliferation of similar risk patterns (i.e. they only differ in the possible paths between the compromised asset and the sensitive data). Therefore, the level of detail of the meta-model constitutes an important trade-off between the accuracy of modeling cloud configurations and the effort for modeling the risk patterns.

7 Related Work

We discuss the work most relevant to ours along two aspects: (1) data protection risks of cloud services and (2) model-based approaches for cloud security and privacy.
7.1 Data Protection Risks of Cloud Services

Risk management covers the process of describing, detecting and mitigating risks. So far, only few frameworks for risk management of services have been presented [18].

Djemame et al. [8] propose a risk assessment framework for cloud computing which is designed to help cloud users and cloud providers assess risks during service deployment and operation. This approach focuses on the relationship between service providers and services. However, they do not state how risks may be monitored during operations. This is where risk patterns can help by specifying what cloud configurations to look for during operations to determine risky situations.

Meszaros and Buchalcevova [18] present a framework for online service risk management. They consider similar assets to ours and present a risk and threat model as basis. They focus on risk assessment and mitigation and propose techniques for risk monitoring. Our approach can be considered complementary to their work as our risk patterns capture specific configurations of cloud services and systems that would lead to high data protection risks.

Several authors have analyzed specific data protection risks in the context of cloud computing and services. Paquette et al. [20] analyzed the risks of cloud computing, focusing on the context of governmental use of cloud computing. Fernandes et al. [10] surveyed security issues in cloud computing as a potential source for data protection risks. These insights provide an important source of input for our approach as they help defining and specifying risk patterns by taking important data protection concerns into account. Our approach can be seen as a vehicle for capturing and utilizing this kind of knowledge.

7.2 Model-Based Approaches for Cloud Security and Privacy

Apvrille and Roudier introduced Attack Graphs based on the SysML-Sec framework [2]. Similar to risk patterns, also attack graphs are visual representations of security threats. However, attack graphs are used to model malicious attacks on – especially embedded – systems, whereas risk patterns encode cloud configurations that can potentially lead to data protection issues. That is, an attack graph models all the details of an attack, including the tools used and activities performed by an attacker, whereas a risk pattern models only the cloud configuration that could potentially be exploited, thereby enabling more general preventive measures. Attack graphs were applied to model a single attack, whereas we compiled a catalog of 45 risk patterns.

Watson and Little [31] introduced an approach to reason about the deployment of a distributed system and its impact on security. They state that not all deployment problems can be solved during design time, so run-time reasoning is needed. In contrast to our risk patterns, their approach requires the assignment of security levels to all assets, which can be difficult in some settings. In fact, risk patterns could help here: the number of risk patterns in which a given asset type appears may be used as an indication of the security requirements of that asset.
Beyond the types of entities considered in that paper, we also explicitly consider actors. As shown in our paper, actors are important for accurately determining data protection concerns.

Similarly to our approach, the work of Schmieders et al. also applied model-based adaptive methods to data protection in the cloud [24, 25]. That work, however, is limited to one specific type of privacy goals: geo-location constraints. Our work, in contrast, addresses data protection goals in a much broader sense.

Kritikos and Massonet proposed a domain-specific modeling language for modeling security aspects in cloud computing [14]. This includes security controls, security properties, security metrics, and security capabilities. In contrast, our work focuses on modeling the typical assets of cloud systems and their relationships, which are the possible attack surfaces and make up the configurations that may lead to data protection violations.

8 Conclusions and Future Work

In this paper, a catalog of risk patterns was elaborated on the basis of vulnerabilities described in the literature. The results demonstrate that our previously proposed risk pattern approach [26] in combination with the meta-model described in [17] is capable of modeling typical data protection vulnerabilities. The present work also sheds light on the limits of the applicability of the risk pattern approach and the typical characteristics of risk patterns.

Several directions for future work remain. First, the syntax and semantics of the language of risk patterns should be defined formally. The catalog of risk patterns elaborated in this work is an important input for the formal definition of the language, as it shows the different constructs that must be supported. Second, the process of devising risk patterns should be formalized based on the experience reported here, and then validated by using it to create further risk patterns. Third, an efficient algorithm should be devised and implemented to search for risk patterns in the cloud model, using the gained insights about the structure and size of risk patterns. Fourth, it should be investigated how the expressive power of the language could be increased by introducing wildcards or other mechanisms to compactly represent families of related risk patterns.

Acknowledgment. This work was partially supported by the European Union’s Horizon 2020 research and innovation programme under grant 731678 (RestAssured).

Appendix

The following picture shows the underlying meta-model (without the packages ‘Goals & Metrics’ and ‘Mechanisms’):
References