Security for Virtualized Distributed Systems : from Modelization to Deployment

Arnaud Lefray

THÈSE

en vue de l'obtention du grade de

Docteur de l'Université de Lyon, délivré par École Normale Supérieure de Lyon

Discipline : Informatique

Laboratoire de l'Informatique du Parallélisme

École Doctorale en Informatique et Mathématique de Lyon

présentée et soutenue publiquement le 3 novembre 2015 par

M. Arnaud LEFRAY

Security for Virtualized Distributed Systems: From Modelization to Deployment

Sécurité des Systèmes Distribués Virtualisés : De la Modélisation au Déploiement

Directeurs :	M.	Eddy	CARON
	M.	Christian	TOINARD
Après l'avis de :	M.	Nicolas	ANCIAUX
	M.	Gilles	DEQUEN

Devant le jury composé de :

M.	Nicolas	ANCIAUX	*Inria Paris-Rocquencourt*	Rapporteur
M.	Eddy	CARON	*École Normale Supérieur de Lyon*	Directeur
M.	Gilles	DEQUEN	*Université de Picardie Jules Verne*	Rapporteur
M.	Marc	LACOSTE	*Orange Labs*	Membre
Mme.	Christine	MORIN	*Inria Irisa Rennes*	Membre
M.	Jean-Marc	PIERSON	*Université Toulouse 3 Paul Sabbatier*	Membre
M.	Jonathan	ROUZAUD-CORNABAS	*INSA de Lyon*	Co-encadrant
M.	Christian	TOINARD	*INSA Centre Val de Loire*	Directeur

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In our contemporary society, most IT services are going online. Nowadays, any user of a service such Gmail expects to access it from anywhere on earth, without delay, at any time and in a responsive and efficient manner.

To cope with our growing needs, the infrastructures hosting these services have continuously evolved taking the shape of distributed systems, the most recent being the Cloud computing. The idea behind Clouds is simple: mutualize the resources to decrease the cost and mutualize the expertise to propose robust platforms offering on-demand resources. The key technology enabling these concepts is virtualization. It gives the impression to a user of having dedicated resources where in fact the underlying real resources are shared amongst several independent users.

Cloud providers offer many promises such as highly available platform (*e.g.,* up to 99.999999999% over a year for Amazon S3 [1]). But one fundamental aspect is left behind this gigantic trend: security. And this negligence is a bargain for hackers. In 2009, numerous systems of the major Cloud provider Amazon were hijacked to run Zeus botnet nodes. In April 2010, the same Amazon experienced a Cross-Site Scripting (XSS) bug that allowed attackers to hijack credentials from the site. On April 20, 2011, Sony Playstation Network, which is Cloud-based, was break down and nearly 77 million accounts were compromised due to an external intrusion few days before and the network was operational again only on May 1. These examples concern major stakeholders which should have the means to protect themselves. With similar stories covering national newspapers, people start realizing that security is critical.

Despite that virtualization has slightly improved security by providing a sense of isolation, many studies have demonstrated some flaws in virtualization that allow the exfiltration of critical information. Moreover, if we look at current security practices, we will find out that security configurations are usually done by hand with all the problems it implies: human errors, gap between the user requirements and the understanding of the security expert, lack of adaptability, and absence of scalability.

Nevertheless, many companies outsource their applications/services to these virtualization-based infrastructures. However, the provider security does not always match tenants' concerns. For instance, Amazon Web Services (AWS) have the following terms of use:

[1] `https://aws.amazon.com/s3/details/`

> *"AWS products that fall into the well-understood category of Infrastructure as a Service (IaaS) – such as Amazon EC2, Amazon VPC, and Amazon S3 – are completely under your control and require you to perform all of the necessary security configuration and management tasks. For example, for EC2 instances, you're responsible for management of the guest OS (including updates and security patches), any application software or utilities you install on the instances, and the configuration of the AWS-provided firewall (called a security group) on each instance. These are basically the same security tasks that you're used to performing no matter where your servers are located."*

In short, the tenant has to do the same effort as for on-premise infrastructures to secure its outsourced business. Meanwhile, because the massive adoption of Clouds makes them a much more attractive target for hackers, the tenant's application is more at risk than before. Moreover, in case an application is compromised and it is determined that the source of the infection is a weakly secured component, the tenant of the last may be held responsible.

In this Thesis, our goal is to provide an end-to-end security of virtualized distributed systems: from the user modelization to the deployment of its application and security.

1.1 Motivations

This Thesis revolves around two ideas. The first idea is to follow a user-centric approach which is an alternative to the *security by default* or *by design* model usually proposed by most providers and many researchers. We are interested in the security of the end-user application and not in the security of the provider platform. The second idea is to bridge the gap between the user's ability to specify security requirements and complex configurations of security mechanisms.

We believe that virtualized distributed systems such as Clouds are to become massively adopted in the near future. It follows that our first goal is to offer an end-to-end security for real applications deployed on such distributed systems. Moreover, a security is relevant only for the application it protects *i.e.,* each application may have different security requirements. Therefore, a second goal is to facilitate the expression of the users' requirements for both the application and its security. Also, the security offered to the end-user should be supported by strong guarantees. A strong security cannot be a patchwork of mechanisms configured by hand. Moreover, the end-user should not manage the complexity of the underlying middleware and related security services. Instead, our third goal is to provide an automatic deployment of the application and automatic enforcement of its security with a formal basis to demonstrate a safe enforcement. Our solution must be as flexible and generic as possible to encompass a wide range of security mechanisms.

In the next sections, we discuss security foundations and targeted infrastructures: distributed systems. First, we explain what is *security* in computer sciences, what does this term cover and what kind of security we are interested in. Then, we review the evolution of computing infrastructures and highlight their specificities *i.e.,* what makes their security different.

1.2 Information Security

Information security means protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction. An **information system** (or system for short) is the combination of people and computers processing or interpreting information. Information security is a general term that can be used regardless of the form the data may take (*e.g.,* electronic, physical). When applied to computing devices, the term used is **Cybersecurity** (or Information Technology security). Cybersecurity may include physical security *i.e.,* preventing theft of equipments. In this Thesis, our concern is information security on digital data against logical threats *i.e.,* attacks conducted using software tools. Accordingly, protecting physical access to equipments is out of scope. In the rest of this document, the term **Security** is used for information security on digital data.

In the rest of this section, we lay the basis of security and determine our scope of interest for this Thesis.

1.2.1 Security Policy

The definition of *what it means to be secure* is expressed by the **Security Policy**. A policy constrains *actions* between *entities* in a system. An **action** is any operation changing the state of the system when performed *e.g.,* reading a file, sending a packet over the network, modifying permissions, log in. An **entity** is anything existing concretely or abstractly regardless of its nature or complexity *e.g.,* a user, a process, a piece of data, a network.

Central to security is the protection of **assets**. An asset is anything of value either substantial like data, computing resources, storage space but also intangible like reputation. For example, disrupting access to service will damage the reputation of the brand and users will be more willing to go for a competitor if it provides a more reliable service.

A key concept of security is the notion of authorization. A security policy can be seen as a set of authorization or denial rules. Enforcing these rules requires preliminary steps. The first step of a security process is **identification**: assigning an identity to an entity. For example, an individual (Bob) wants to attend a conference, he declines his identity by telling his name. Because his word should not be taken for granted, his identity must be verified: this is the **authentication**. Therefore, the conference organizer authenticates Bob by asking for his ID card. Next, an authorization is delivered based on a **control** mechanism. The conference organizer checks if Bob is on the participants' list before allowing him to attend the event. Even if the first 2 steps are mandatory, they are already well covered and solutions exist even for distributed systems. Thus, in this Thesis, we will focus on the control mechanisms itself *i.e.,* the authorization step.

1.2.2 Security Properties

A security policy may be composed of basic rules defining what it is to be secure. Nevertheless, subsets of rules may enforce more general objectives called **security properties**. For example, you may install sensors and alarms in your house. Your rules are to activate the sensors whenever you are away from home and trigger the alarms if someone is detected by a sensor. But your general objective *i.e.,* the property you want to hold, is to be protected from theft and your installation is a mean to enforce this property. A security policy may be directly stated as a set of security properties.

In security, three main concepts commonly known as the CIA-triad (not to be confused with the US agency) has been widely used for decades: **Confidentiality**, **Integrity** and **Availability**. Both the Department Of Defense guidelines (TCSEC/Orange Book) [115] edited in 1985 and the more recent *Common Criteria* (ISO/IEC 15408) international standard define security as an integration of availability, confidentiality and integrity. They define the three as follows.

Confidentiality

It is the *absence of unauthorized disclosure of information.* Often companies have internal confidential documents for the sole use of agents with appropriate clearance. These documents can contain trade-secrets or marketing strategies and their public disclosure may result in loss of competitive advantages or loss of reputation. It should not be confused with privacy which is *the right of individuals to hold information about themselves in secret, free from the knowledge of others* [94]. For example, improper enforcement of confidentiality of healthcare records may arise privacy issues, especially if records are not anonymized.

Integrity

It is the *absence of unauthorized system alteration.* Information has values only if we are sure that it has not been tampered with. For example, if you are transferring $100.00 to a remote account, you do not want the amount to be changed to $10,000.00. Integrity ensures the absence of malicious or unintentional modification or destruction of an asset.

Availability

It is the *absence of unauthorized denial of use.* A service or data is available if any authorized entity accessing it is served in a reasonable timeframe. Availability may be related to *fault-tolerance* and *reliability* [77]. For example, an attacker floods the network to saturate accesses to a website. The website would not be able to serve all users in timely manner due to the huge proportion of dummy requests. The attacker has disrupted the ability of the website to deliver its service in the *expected* timeframe. In particular, he has not necessarily read or modified any unauthorized data but he has prevented an authorized user to do so and thus has broken the availability property.

Isolation

It is the *absence of unauthorized disclosure and/or modification of information.* It is a composition of integrity and confidentiality properties. In the general sense, isolation is used to refer to the absence of any possible interactions (except intended ones *e.g.,* public interfaces) between two entities. As a straightforward example, isolating two antagonistic persons may be done by putting them into two separate rooms. This idea is used for isolating applications from each other or from the rest of the system.

1.2.3 Mandatory and Discretionary Control

The enforcement of security properties depends on the environment they apply to. At first, information security was a critical matter for the military. Typically, information was classified into *top-secret*, *secret* or *public* domains and an agent with top-secret clearance could access all three domains whereas an agent with public clearance would be limited to accessing public domain information. These constraints on the information are part of the security model. Nowadays, information security is critical to everyone and these models are not suitable outside the military domain.

A **Security Model** specifies the entities of the system, the classification/hierarchization of entities, how permissions are set or may be updated. It usually focuses on one or few security properties.

A **Control Model** defines *who* is in charge of delivering permissions.

The most widely used control model is the **Discretionary Control** model where controls on access to an object may be changed by the owner of the object itself. For example, in the traditional read-write-execute GNU/Linux permissions, a file's owner may change its permissions as he wishes. Moreover, the privileged (root) user is allowed to update permissions for any object in the system. Consequently, a juicy situation for an attacker is to obtain the root identifier. Furthermore, as human configuration of security permissions is error-prone, it often leads to globally insecure systems.

In opposition to the discretionary control model is the **Mandatory Control** model. In it, a third-party distinct from any user of the system (privileged or not) regulates the set of permissions. Consequently, even the most privileged user must follow the security policy and has no ability to modify it. For instance, SELinux [83] is a mandatory *access control* implementation on Linux-based systems developed by the NSA where even the root user may be constrained by its permissions. Originally, SELinux was an implementation of the Flask operating system security architecture [112] which focuses on providing an administratively-defined security.

1.2.4 Access Control

In an **Access Control** (AC) model, permissions about accesses to entities are explicitly stated. Typically, the question you ask is: "Can I read this file?" and the answer depends on your identity, your affiliations (*e.g.,* groups, domains), for example: "Because you are an administrator, access is granted to you". Access control models may be classified as discretionary or mandatory.

Lampson The first Discretionary Access Control (DAC) model was formalized by Lampson in 1971 [76]. He divided entities into *active* and *passive* entities, respectively called *subjects* and *objects*. A subject is typically a user or a running process whereas an object is a file or a binary code. Subjects are grouped into *domains*. He proposed an access matrix with domains in rows and objects in columns to define permissions. Lampson's model was improved by Harrison *et al.* [59] with the HRU model introducing generic permissions (*e.g.,* own, read, write, execute), system commands (*e.g.,* create file, update rights), and actions to add/delete permissions/entities. Harrison *et al.* demonstrated that, in the general case, *verifying a discretionary control model is undecidable.* This proof is essential to discard discretionary approaches for guaranteed security properties.

Bell-Lapadula The Bell-Lapadula model [12] focus on data confidentiality in a military context. It is a Mandatory Access Control (MAC) model. It relies on the concept of *state machine* where all possible states of the system are divided into *secure* and *insecure* states. Consequently, It is proven that a secure system can only transit from a secure state to another secure state. A state is *secure* when permissions of subjects to objects comply with the security policy that includes the following main security properties:

- A subject at a given security level may not read an object at a higher security level (no read-up).
- A subject at a given security level must not write to any object at a lower security level (no write-down).

Bell-Lapadula model only applies to multilevel security systems based on clearance or need-to-know principles. As a result, it is not suited for any collaborative/competitive structure not representable by a lattice. Furthermore, without declassification process, the data tends to go up the security clearance level without any chance to go down.

Biba The Biba model [16] is a MAC model focusing on data integrity. It is the direct inverse of Bell-Lapadula confidentiality model adapting *no-read-up* and *no-read-down* properties into *no-write-up* and *no-write-down*. Again, it only applied to multilevel security systems.

Role-based Access Control In previous models, permissions are set for each user and users may be attached to groups. It was noticed that computing permissions or group membership does not scale in a decentralized model (*e.g.,* discretionary). For example, in the read-write-execute Linux model, determining the right for accessing a file requires computing the traversal of all directories. To support larger systems, **Role-based Access Control** (RBAC) was proposed [108]. Instead of having groups of users, users have roles. A major difference between groups and roles is that groups are typically treated as a collection of users but not as a collection of permissions. A role, serving as an intermediary, is both a collection of users and a collection of permissions. Role-based access control model tackles *delegation* [118]: allowing a lower-level user to do an action in the upper-level user stead. For example, the *grant* permission is implemented in most databases to add a permission to users. OASIS [4] is a RBAC architecture for secure interoperation of services in an open, distributed systems.

RBAC eases the specification and administration of security by structuring entities around roles and emphasizing on delegation of rights, but it does not improve security.

1.2.5 Information Flow Control

In an **Information Flow Control** (IFC) model, the goal is to understand and control how information flows between entities that is how entities interfere with each others. To the same question “Can I read this file?”, an IFC mechanism will wonder: “It depends on what you and the other entities have done before”.

Information Flow Control and Access Control can be seen as different but complementary approaches to security [67]. Access Control makes explicit statements about permissions for a subject to realize a specific action on a resource whereas information flows are implicit. However, IFC makes explicit statements about permitted information flows whereas permissions for a specific action are implicit.

In short, Access Control checks place restrictions on the release of information *but not its propagation*. Once the information is released from its container, a program may (maliciously or unwillingly) leak the information to an unauthorized entity. Denning [41] performed basic research on IFC in 1976. He defined the concept of an information flow policy as a triple $< SC, \rightarrow, \oplus >$ where:

- SC is a set of security classes.
- $\rightarrow \subseteq SC \times SC$ is a binary *may-flow* relation.
- $\oplus : SC \times SC \rightarrow SC$ is a binary class-combining or join operator on SC.

An important property in IFC research is **non-interference** [51]. It means that a group of users, using a certain set of commands will not interfere with another group of users if what the first group does with those commands has no effect on what the second group of users can see. This property demonstrates the key difference with access control by looking at effects and consequences rather than permissions.

IFC allows enforcing *end-to-end* security policies: from the place information originates to leaving the system [107]. It is studied at two levels: language and system.

Language-based Information Flow Control

A **Language-based IFC** focuses on flows *inside* a program, in particular between variables [107]. Flows can be explicit or implicit. Suppose the following programs with 2 variables (`secret` and `public`):

```
public := secret
```

An *explicit flow* is created between the two: `public` contains the `secret` value.

```
if secret then
  public := 1
```

An *implicit flow* is created between `secret` and `public`: inspecting the value of `public` allows inferring the boolean value of `secret`.

It is particularly important for a cryptographic library not to expose critical information such as secret keys used to encode data. For example, in [25], Brumley *et al.* has shown that extracting private AES keys from a remote client based on OpenSSL implementation is practical.

One direction envisioned to tackle explicit and implicit flows is *Security-Type Systems* [111]. In this approach, the types of program variables and expressions are augmented with annotations that specify policies on the use of the typed data. Then, information flows are statically checked at compile-time to guarantee that no paths exhibit an insecure flow. In [90], Myers proposes JFlow: an extension to the Java language permitting static checking of flow annotations. For example, to the previous *implicit flow*, he proposes to attach labels to types and he obtains the following solution:

```
int{public} x;
boolean{secret} y;

if y then
  x := 1
```

Language-based solutions cannot cope with *system-wide* security which is essential to tackle large systems involving multiple organizations.

System-based Information Flow Control

A **System-based IFC** focuses on controlling flows of a system. The goal is to track and control how information flows between entities *e.g.,* processes, files, sockets. The approach is less invasive than language-based IFC *i.e.,* it does not require any modifications of the application. However, it comes with a loss of granularity and cannot cope with the previous example of cryptographic library leaks.

System-based IFC has been popular for building **Intrusion Detection Systems** (IDS) where the objective is to monitor activities in a system and detect potential intrusion by a malicious user/program [62, 99, 126, 132]. PigaOS [23] is a context-based IFC system originally designed as an IDS supervising indirect flows within a SELinux policy but it has evolved into a control mechanism later on. The complexity of writing a policy often leads PigaOS to detect false positives (*i.e.,* normal flows detected as malicious).

Most models relies on **Tainting** techniques. The idea is to attach colors to entities and to propagate them whenever an interaction is made [36]. For example, if a sensitive file is tainted in red and as a user you read it, you will inherit the red color. If now you write another non-sensitive file, it will be contaminated by the red color too. In [44], Enck *et al.* propose TaintDroid: a system information-flow tracking system for realtime privacy monitoring on smartphones. Information is tracked at multiple-levels: variable-level, method-level and file-level. The process itself is not purely system-based: variables and methods are tracked. But this is only possible for a system based on a single language *e.g.,* Java for Android.

Decentralized Information Flow Control

A **Decentralized Information Flow Control** (DIFC) is an approach to security that allows application writers to control how data flows between the components of an application and the outside world [73]. In a DIFC approach, part of the security is managed independently by multiple components.

In [91], Myers proposed JIF (previously JFlow), a language-based DIFC dedicated to privacy protection in Java. JIF is based on *decentralized labels* where labels are attached to variables. Example of a label is $\{o_1 : r_1, r_2; o_2 : r_2, r_3\}$ where o_1, o_2 are owners and r_1, r_2, r_3 readers. Owners and readers are *principals*: groups or roles composed of entities, and principals are ordered under a lattice. The particularity of decentralized labels is that a principal may choose to selectively declassify its own data, but does not have the ability to weaken the policies of other principals.

The decentralized labels of JIF are reused in AsbestOS [43], HiStar [127] (which is based on AsbestOS) and Flume [73] at the system level. AsbestOS and HiStar implement labels as a combination of categories (similar to principals) and security levels. Flume uses secrecy and integrity as distinct labels.

All these works focus on a limited set of security properties (*i.e.,* confidentiality and/or integrity) in multi-level security systems.

Access Control and Information Flow Control

It is not easy to see if an AC policy can leak information to a potential attacker and many studies have been led in this direction, namely AC policy analysis [14]. In [54], Guttman *et al.* use model checking to determine whether information flow security goals hold in a system running SELinux. Information flow statements are expressed in Linear Time Logic [98] *i.e.,* a logic with temporal modalities where formulas are statements about the future of executions *e.g.,* , a condition will be true or will be after another fact becomes true. Similarly, Hicks *et al.* [61] check SELinux policies against information flow properties formalized as a logic implemented in Prolog. Nevertheless, this analysis process is done offline and must be redone every time the policy changes. Accordingly, it is impossible to do it on a dynamic system where some actions may impact the policy (*e.g.,* creating or deleting a user).

At the opposite, in [68], Kahley *et al.* factorize high-level information flow specifications into low-level access controls. Their model is limited to confidentiality and integrity labels to cope with composable specifications.

Discussions

Access control approaches are easy to verify and thus incur low overhead. But they lack the ability to control the propagation of information in a system which is important to guarantee *end-to-end* security. Static analysis of access controls may permit verifying information flow properties but it cannot cope with dynamic systems where users, groups, roles are created or removed at runtime (*i.e.,* elasticity is out of the scope of this Thesis).

Most AC or IFC approaches tackle small sets of properties, usually confidentiality and integrity (or both). Adapting them to integrate more flexible properties seems at least difficult. Most models are based on multi-level security which we believe is too rigid to represent multi-organizational systems where security levels alone have no sense.

Languaged-based IFC and system-based IFC can be seen as complementary. The first is suitable to control flows between programs' variables and can deny implicit flows but to provide system-wide security the second is more appropriate. The downside of IFC is the computation complexity graving performances.

Most of the literature tends to associate discretionary or mandatory terms to access control and neglects the fact that information flows can also be controlled in a discretionary or mandatory fashion. Discretionary approaches cannot provide strong, provable security guarantees despite their are widely available and somehow necessary.

In this Thesis, we mainly consider on **Mandatory Information Flow Control at system level** and see it not only as a standalone security mechanism but also as a way to configure other security mechanisms such as access controls, firewalls, cryptographic protocols and placement algorithms. That is why the considered formal logic mainly deals with Information Flows. However, that formal logic enables to automatically compute or configure the underlying security mechanisms even if mandatory mechanisms are not available.

1.3 Distributed Systems

Distributed Systems [52] are groups of interconnected computers. Physical machines (computers) are *distributed* across different locations from the room next door to the other side of the globe and communicate through a network. Each machine has its own local storage and processing power and shares information with other machines to complete a common objective. Machines may collaborate to process a complex computational-intensive task such as weather forecast. Each machine may deliver more *lightweight* services *e.g.,* webmail, data storage to users and thus share resources to diminish costs and/or provide high availability. In a distributed system, each machine is referred as a **node** of the system.

In this section, we focus on major targeted infrastructures and highlights their key characteristics.

1.3.1 Clusters

A **Cluster** consists of a set of loosely or tightly connected computers working together. Initially, the idea was to interconnect commodity hardware to build a cost-efficient distributed system at a time when a single computer was not powerful enough to deal with users' needs. Clusters exist in all possible scales: from two to thousands of nodes.

Nowadays, web giants build and maintain large Clusters to sustain users growing demand in high quality services processing massive data sizes. For example, in 2003, Google [7] reported combining more than 15,000 commodity-class computers in multiple Clusters in order to handle web search queries all over the world. Their high-throughput solution is more cost-effective than a comparable system built out of a smaller number of high-end servers such as mainframes.

In private sector, a Cluster is typically for the sole use of a single company (*e.g.,* Google, Facebook, Yahoo) and must be secured against external threats: cyber-attacks conducted through Internet.

1.3.2 Grids

As Clusters are very popular and widely adopted by many organizations, a natural question is whether it is possible to federate the resources of organization to build an even more powerful distributed system. **Grids** provide a solution to this issue. The term originated as a metaphor: any user should be able to access computing resources as easily as on US electrical power grid and in a standard fashion. In [50], Foster *et al.* define that Grids solve the problem of coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations.

The key shift is the *virtual organizations* characteristic. In [49], Foster *et al.* discuss a security architecture for Grids. They devise a security policy integrating an heterogeneous collection of locally administered users and resources but fall short in sound and scalable solutions. In particular, the question of interoperability between local access control policies is left open.

In practice, Grids in production are mostly scientific national or international platforms where users are supposed to be honest *i.e.,* no intentional misbehavior. Therefore, security are mainly strengthened for authentication and credentials [27, 49, 66, 101, 125]. The Grid is an highly heterogeneous platform from the hardware infrastructure, the middleware to provision resources, and the security managed locally by each organization. In such environment, providing strong security guarantees over the whole Grid is more than difficult.

1.3.3 Clouds

The concept of **Cloud Computing** [128] goes back to 1960 with the introduction of Utility as a service [95] by John McCarthy speaking at the MIT Centennial. The goal was to bill computing power as water or electricity. The computing power would have been provided by a large infrastructure such as a shared Grid. In the past, building such infrastructure was impossible due to high costs and the lack of network efficiency. However, nowadays with affordable and efficient hardware, it has become possible.

The mainly considered characteristics are:

- *On-demand resources.* Users can use computing or storage resources as needed automatically without requiring human interaction with the service provider.
- *Elasticity as dynamic scaling.* Users can automatically adjust their resources upward or downward depending on the current workload hence reducing costs.
- *Multi-tenants provisioning.* Utilization of resources is maximized by sharing them between multiple tenants.
- *Pay-as-you-go.* Most widely known economical model, users pay per time slice ($\tilde{1}$ hour) of provisioning.

Cloud Computing enables cost reductions due to resource sharing. Instead of investing into on-premises infrastructures they have to manage, maintain and secure, companies can take advantage of Cloud elasticity and scalability to dynamically adjust (and pay) their resources according to their business. For example, an online tickets shop has a peak demand during the few days whenever an event is open for tickets sale. Cloud Computing is well-suited and promising in such cases. Without, it would have bought and managed resources according to the peak thus having an overdimensioned infrastructure for normal periods.

Four organizational models have been identified for Cloud architectures:

- *Public.* The service is available to the general public and managed by a third party.
- *Private.* The service is dedicated to a company but may be managed by a third party.
- *Community.* The service is shared between a common community composed of multiple parties and may be managed by a third party.

- *Hybrid.* The service is orchestrated between at least two Cloud models (public, private or community). Usually, an organization provides and manages some resources in-house (private) and has others provided externally (public).

Cloud computing is usually divided into three service models. Figure 1.1 illustrates the differences between each model and in respect to on-premises infrastructures where the provider manages the entire stack.

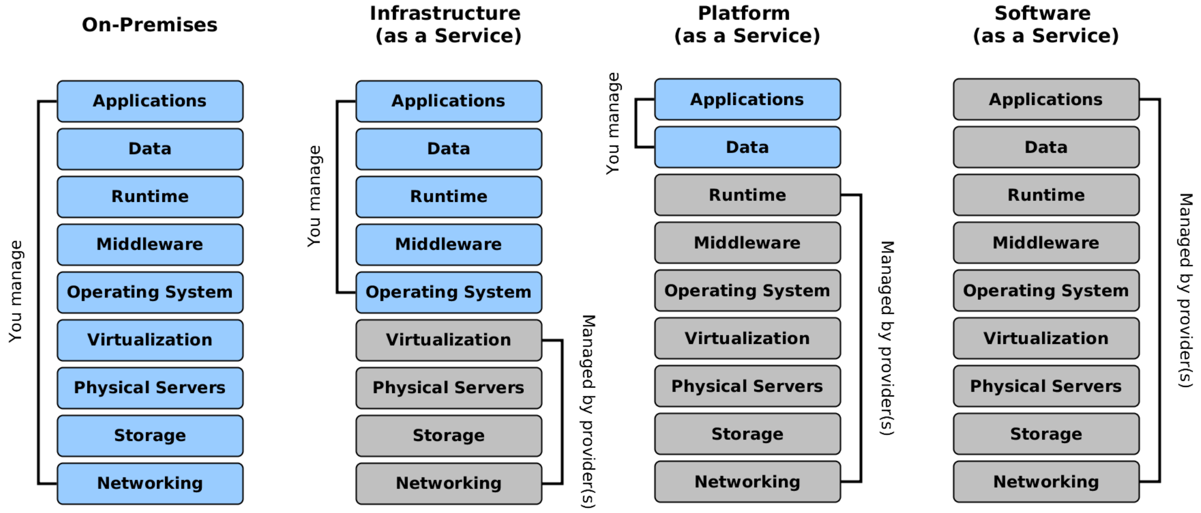


Figure 1.1: Cloud Stacks and Separation of Duties

Infrastructure as a Service (IaaS) Virtual resources such as computation, storage or network resources are offered as a service. For example, Amazon Web Services (EC2 for the computing and S3 for the storage), Microsoft Azure provide IaaS solutions.

Platform as a Service (PaaS) A computing platform is offered as a service. It includes execution runtimes, databases, web servers. Google App Engine and Salesforce provide PaaS solutions.

Service as a Service (SaaS) Also referred to as on-demand software, it proposes services for end-users. For example, Google Drive and Dropbox provide SaaS solutions.

The main enabling technology for Cloud computing is **virtualization**. It allows offering homogeneous virtual resources to the tenant on top of heterogeneous physical resources. Shared resources are efficiently managed and, to some extent, isolate tenants between each other. We detail virtualization techniques in Section 1.4.

Federated Clouds

A single Cloud provider is bounded in number of clients. If he wants to sustain a huge peak demand, he must invest in a larger infrastructure at the risk of overprovisioning, otherwise he will be forced to refuse new clients at the risk of loosing reputation and potential business. A profit-oriented approach consists in building a **Cloud federation** [32]. In this approach, the benefit is mutual: a saturated provider could delegate extra requests to other providers; the first keeps satisfying the demand and the second generates new revenues from its underused infrastructure. Federated Clouds are also supposed to provide users with better quality of service.

Federated Clouds face many obstacles. Because Clouds provider have different architectures, technological choices, pricing models, etc., building a unified interoperable federation is a challenge. Moreover, how can one be sure that each provider of the federation is able to ensure the same security guarantees.

Cloud Security

In a 2015 security survey in collaboration with 250,000+ contributors[2], security is still the biggest perceived barrier to further Cloud adoption. Yet, 22% of the contributors have experienced *less* security breaches in public Clouds compared to on-premise whereas 28% have experienced *more* breaches. This result could be explained as small and medium-sized enterprises cannot afford costly security solutions for their in-house infrastructures. As a result, for them Clouds concentrate security expertise and thus improve their overall security even if Clouds providers secure *their* infrastructures and *not* what is deployed (*e.g.,* VMs, applications, data) on top.

However, Clouds present a real challenge to security researchers [133]. With the augmentation of parties, threats of data compromises increase as well. Outsourcing data in the Cloud is similar to delegating control over data to the provider. Then, the question is how much do you *trust* your provider. Furthermore, multitenancy is at several levels including hardware: an attacker may provision resources to sit next to his victim and conduct exploits.

In short, Clouds must face both external threats (as on-premise data centers do) and internal threats due to multitenancy.

In Clouds, resources provisioning is opaque. It means the provider may arbitrarily provision resources located in the US or in Europe. However, depending on the country the resources are located, the legislation may differ and it matters [96]. The risk exists that personal data may be used for profiling using data mining techniques or that business secrets are revealed to foreign competitors. For example, nothing forbid the US government to take a look at your data if you are not a US citizen and the problem is that major public Cloud providers have their infrastructures located in the US.

1.4 Virtualization

Virtualization consists in creating a virtual (rather than actual) version of computing components. It includes (but not limited to) hardware components (*e.g.,* CPU, memory, devices), networks, operating systems, file systems, databases and applications. Virtualization is used to multiplex a single resource into multiple virtual ones. Accordingly, it allows the sharing of physical resources while virtual resources seem to be dedicated.

Hardware Virtualization is the virtualization of hardware components and operating systems. For example, virtualization allows for running Windows OSes on top of a

[2] `http://www.infosecbuddy.com/download-cloud-security-report/`

Linux-based OS.

Figure 1.2 illustrates the principle of hardware virtualization. Compared to non-virtualized systems, the operating system is replaced by an **Hypervisor** (or Virtual Machine Monitor) which role is to manage and run **Virtual Machine** (VM). A virtual machine is a virtual container for an operating system called *Guest OS* in opposition to the hypervisor sometimes referred as the *Host OS*.

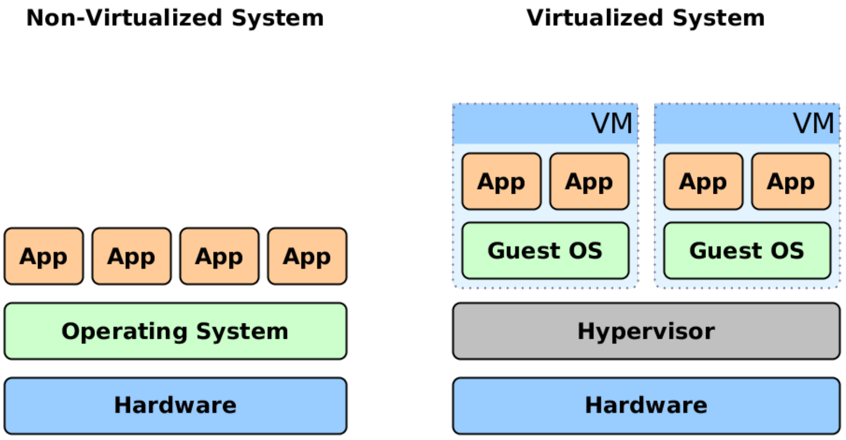


Figure 1.2: Non-virtualized and virtualized systems

Multiple solutions have been designed to implement hardware virtualization.

1.4.1 Full Virtualization

In **Full Virtualization**, the hardware is almost completely simulated to allow the guest OS to run *unmodified*. The guest OS is not aware it is being virtualized and it does not require any modifications to run correctly.

Because guest OS and hardware are completely decoupled, full virtualization provides good security and performance isolation for virtual machines. Furthermore, *migration* (*i.e.*, moving the guest OS from one hypervisor to another) is simple and *portability* is ensured: the same OS can run in a virtual machine or on a native hardware.

A key challenge for full virtualization is the interception and simulation of privileged operations, such as I/O instructions. A privileged operation has the ability to modify the state of other virtual machines. To keep privileged operations contained, *binary translation* [104] techniques analyze the guestOS binary code to trap such operations and replace them with safe equivalent instruction sequences. The downside of binary translation is the high overhead it incurs.

Existing full virtualization technologies include Parallels, VMware, VirtualBox, Qemu/KVM.

Hardware-assisted Virtualization

The concept of **hardware-assisted virtualization** first appears on the IBM System/370 in 1972. To improve full virtualization (but also paravirtualization) efficiency and in particular overcome privileged instructions issues, hardware manufacturers have developed new processor extensions to specifically support virtualization. Intel and AMD have independently designed *Intel VT-x* and *AMD-V* for virtualization on x86 architectures. For instance, they both allow for direct access to I/O devices bypassing the hypervisor.

1.4.2 Paravirtualization

In **Paravirtualization**, the guest OS is offered a software virtualization interface to run. Therefore, the hardware is not simulated, instead the guest OS needs to be ported for the paravirtualization API (Application Program Interface). The value proposition of paravirtualization is in lower virtualization overhead: some instructions are non-virtualizable thus providing an API makes their execution by the hypervisor much more efficient. Hence, non-virtualizable instructions are replaced by *hypercalls*. Critical kernel operations (*i.e.,* memory management, interrupt handling and time keeping) can also be managed through hypercalls.

In resume, a conventional OS distribution that is not paravirtualization-aware cannot be run on top of a paravirtualizing hypervisor. Paravirtualized OSes have better performance compared to full virtualization and also ensure isolated executions.

Existing paravirtualization technologies includes Xen and VMware.

1.4.3 Operating-system-level Virtualization

Operating-system-level virtualization is a *shared-kernel* virtualization technique. As such, it is not an hardware-based virtualization. In classic OS architectures, addresses space is divided into one user space and one kernel space. With operating-system-level virtualization, the kernel allows for multiple isolated user-space instances called *containers*. All containers share the same kernel but run as if they were standalone operating systems. Operating-system-level virtualization does not interpose any hypervisor or interface and contained applications run with native performances.

Existing operating-system-level virtualization includes chroot, LXC (linux containers), Solaris Containers, FreeBSD Jails. Docker is a recent yet popular software based on LXC to easily build, package and distribute applications.

Currently, containers are *not recommended* for strong security isolation [102]. As they share the kernel, applications can exploit any vulnerabilities to pierce through the container's boundaries and infect the rest of the system[3]. However, with popularization, more effort will be put to provide additional security features to containers.

1.5 Discussion

Nowadays, most current applications are distributed to either process huge sets of data or offer highly available services. In the mean time, the evolution from Clusters to Grids then to Clouds demonstrate the need for systems supporting multiple organizations with the inherent risk of antagonistic concerns. In these shared environments, security is a key issue: how to store, process or exchange data in a securely manner. Clouds inherit major security concerns from Grids and Clusters plus introduce new ones. These new issues are

[3] `http://opensource.com/business/14/7/docker-security-selinux`

due to massive multitenancy but also to virtualization as a milestone to effectively build Clouds. We believe future systems design will all be about virtualization as it comes with large benefits *e.g.,* flexibility, portability and interoperability.

Therefore, in this Thesis, we tackle any virtualization-based distributed system with multiple organizations sharing the infrastructure.

1.6 Virtualized Application Use Case

We need to have a good understanding of *what* to protect before designing any solution. In this Thesis, we focus first on virtualized static n-tier applications even if our approach can tackle other classes of applications providing some adaptations. By *virtualized*, we suppose that the applications (processes, data, etc.) are packed into virtual machines and deployed on a virtualization-based infrastructure. *Static*, as opposed to dynamic, suppose that the application does not need any scale (up or down) at runtime. Once the application is deployed, it will run indefinitely without changing its resource requirements. Finally, n-tier characterizes an application usually composed of a set of low-level services (*i.e.,* tier) offering a high-level one. The most common n-tier application is the 3-tier web service. It is composed of 3 low-level services: a frontend, a processing service and a backend database. Each of the low-level service can be instantiated multiple times to sustain the load, requiring a load-balancing mechanism to spread it between instances. Figure 1.3 illustrates a 3-tier architecture where the *user* sends requests to a *web server* (*e.g.,* Apache). The request is relayed to an *application server* (*e.g.,* Tomcat) providing a service. Finally, the service stores and retrieves its data from/to a *database server*.

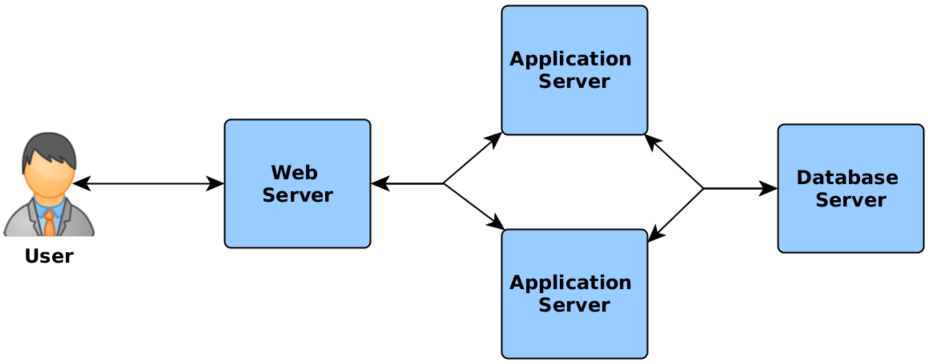


Figure 1.3: n-tier architecture example

This Thesis has been conducted within the Seed4C project [19] where several use cases have been proposed. To better picture the kind of constraints and needs such applications may require, we present thereafter Ikusi's Advertising Content Manager for Airports: a multi-organizational n-tier use case for airport-related content.

1.6.1 Advertising Content Manager for Airports (Ikusi)

The Ikusi company offers a broad range of operational management solutions for smart cities, airports, security, mobility (including railways) and health. In particular, it proposes airport management systems coupled with public information and entertainment systems. These systems manage all airport services such as check-in/boarding, automatic processing of baggage delivery, public information (*i.e.,* gates, announcements) and advertisements. Ikusi solutions are highly mutualized: the company proposes an external management of multiple airports with the same virtualized infrastructure. Airlines send information messages to an airport database application like planes routes, delays, passengers, then these information are exposed to airports operators and devices.

Because Airport management is complex, our use case focuses on the Advertising Content Manager called Musik[4]. This use case is highly multi-tenant with several airports organized in groups. Each group has private information and data exposed by the Content Manager. For instance, employees information must be kept confidential to the airport they work for.

1.7 Contributions

Our problem is how to allow an end-user to specify a secured application and how to automatically enforce and guarantee the required security. In this Thesis, we propose a user-centric approach providing an automatic deployment of virtualized applications with an automatic enforcement of security policies. Our approach is materialized as a toolbox called **Sam4C** (Security-Aware Models for Clouds).

The contribution of this Thesis is to propose an end-to-end framework from an easy-to-use modelization interface for the end-user to the deployment and enforcement of this model. First, we propose a unified model including the virtualized application *i.e.,* the end-user application to deploy, and security requirements *i.e.,* the application security properties to enforce. The idea is to model a virtualized distributed system *i.e.,* a set of interconnected virtual machines, and the associated security policy. This modelization does not rely on any characteristic of the infrastructure. In particular, the user does not know about available security mechanisms. He specifies what he wants and the satisfaction of his requirements, how to implement them, is up to the deployment engine. Using *Sam4C Modeling* (implemented as a standalone Eclipse application[5]), the application can be modeled graphically and the security properties are textually specified with direct references to entities of the application model.

Second, the security enforcement relies on the hosting infrastructure, therefore we propose an infrastructure model *i.e.,* a set of physical machines and networks, and a microarchitectual model *i.e.,* the internal architecture of physical machines. The microarchitecture is used to compute our risk-metric quantifying how much information can leakage between two VMs sharing the same physical machine.

Third, we propose a formalization of the security properties called IF-PLTL. This formalization serves as a proof basis to divide a global property (securing a set of entities) into local typed properties (securing a single specific entity) during a phase called preprocessing. Indeed, existing enforcement solutions enforce properties on specific types of entities (*e.g.,* VMs, Data, Network) whereas a property specified by the end-user protects different types of entities and cannot be enforced directly. Furthermore, we propose a dynamic monitor enforcing any property expressed in IF-PLTL and show that this monitor

[4] `http://www.ikusi.com/en/fids-musik-public-information-system-advertising-content-manager`

[5] `http://www.eclipse.org/`

is effective but at the cost of complexity in time and space.

Fourth, using these models (obtained after preprocessing) as input, we propose a deployment with two different but complementary enforcement solutions: by placement and by automatic configuration. In the placement solution, the objective is to guarantee that the microarchitectural resources (CPU, RAM, etc.) allocated to the end-user satisfy the security policy. Indeed, shared resources may be used as a channel to illegally exfiltrate information. In the automatic configuration solution, *security agents* are dispatched in the virtual machines or at the infrastructure level. The idea is that each agent has a list of properties it knows how to enforce. Indeed, it knows the predefined security tools and policies and can select the right ones automatically. The deployment phase is realized by *Sam4C Scheduler* (implemented as a Java application). *Sam4C Scheduler* has been integrated with an open source Cloud software platform called OpenStack[6].

Fifth, we illustrate our entire end-to-end workflow on a real industrial use case where the modelization has been realized by the company itself.

1.8 Structure of this Document

This Thesis' structure is mapped on our solution architecture depicted in Figure 1.4. In Chapter 2, we present our modelization composed of the virtualized application and the infrastructure detailed in Section 2.2, plus the security requirements detailed in Section 2.3. Then, Chapter 3 presents our IF-PLTL language formally defining security properties. In Chapter 4, we describe the deployment phase. This phase includes first the preprocessing (Section 4.1) which transforms the specified global security requirements into local security properties using formally proved equivalences, then the placement (Section 4.2) which maps VMs to physical hosts while satisfying some security properties, and finally the automatic configuration (Section 4.3) which enforces properties using mechanisms at the VM or host level. Lastly, Chapter 5 run through each previous steps with an Airport Content Manager use case.

[6] `https://www.openstack.org/`

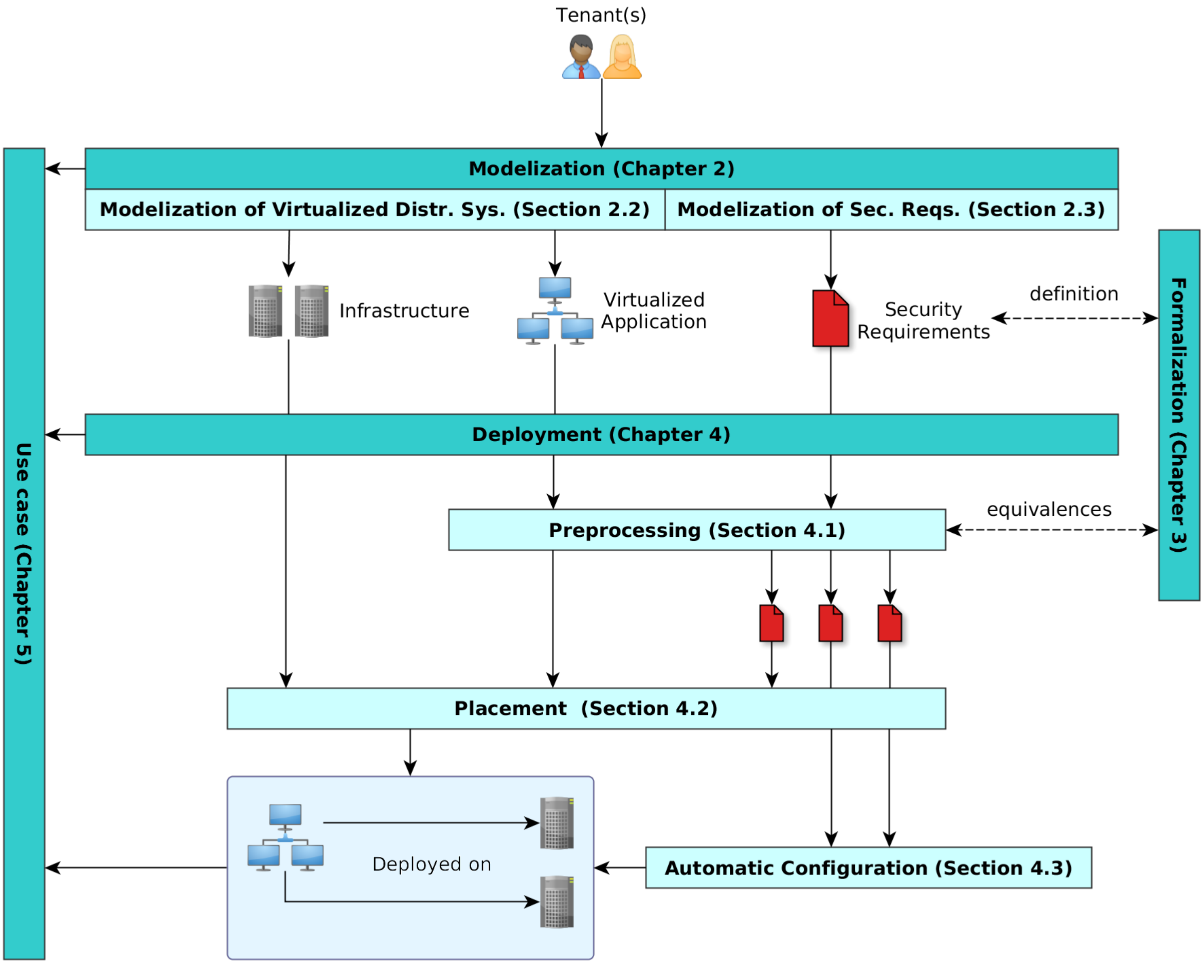


Figure 1.4: Overview of the Thesis

Chapter 2

Modelization of Security Requirements for Virtualized Distributed Systems

In the previous chapter, we have motivated the need to bridge the gap between security enforcement methods and specification requirements. To do so, we need to rely on a description of the characteritics of the application to secure and a description of these security requirements. To construct an house, an architect needs its plans and not the house itself. Similarly, we need a simplified representation (*i.e.,* a model) of the application in order to deploy it and enforce a security relevant to this application. Besides, a model exposes an easy-to-use interface to the user to describe its application and the requirements (including the security).

This chapter discusses existing modelization methods. Using the efficient model-driven engineering approach, we propose a modelization of virtualized distributed systems, their security requirements and infrastructures they are deployed on.

2.1 State of the Art

A **model** is a simplified view of the reality. A model focuses on some aspects of the system it represents. For example, to build a house, an architect will use different plans: some for describing how the building will look like including the dimensions of each room and others to describe accesses to electrical, water and telecommunication networks. In our case, our goal is to describe how the user's application is structured and what security the user needs. The level of details must be sufficient to *automatically* deploy the application and enforce the security requirements. The last model to provide is the infrastructure hosting the application. Indeed, some security issues are due to insecure configurations at the host level. For example, sharing too many resources with a malicious tenant is dangerous. Having a comprehensive infrastructure resource model is critical to control which resources may be shared.

In this section, we review existing works on security models for virtualized distributed systems and show that despite many approaches have been proposed to model applications, security models are mainly limited to RBAC policies and there is little effort to propose a sound and formal global security view of a complex distributed application.

2.1.1 Security Requirements Models

Many models have been proposed to model security requirements (also security properties). In the previous chapter, we have presented the main security models (*e.g.,* access control, information flow) but they are mathematical models which are too complex to be exposed to a user as is (we will discuss them in the next chapter). In this section we are interested in approaches for specifying security properties of a complex application in a user-friendly fashion with automatic enforcement procedures whenever possible.

Security Level-Agreements

One trend is to consider security as a service which can be qualified *i.e.,* measured and evaluated.

A **Service-Level Agreement** (SLA) [69] is a service contract between service provider and service user. A SLA is composed of a list of **Service-Level Objectives** (or Service-Level Target) where each objective is a quality-of-service value for a specific characteristic. For example, a user may require a minimum network bandwidth measuring in Mbps (Mega bits per second). As a SLA is a contract, it also mentions the responsibilities of the involved parties (*i.e.,* penalties), especially in case the provider fails to delivery the expected level of service. If the approach sounds compelling for security, one major issue is how to evaluate the quality of protection and know whether the objectives are fulfilled.

In [60], Henning discusses the possibility of establishing a security SLA for enterprises. His approach is based on standards and recommendations and focuses on contingency planning and user training. For example, one service requirement is *Security violations introduced by individuals due to oversight or intentional introduction to the systems* and it is measured as *a percentage of security incidents introduced by users within target systems*. Then, the best quality of service is achieved if there is less than 2% of security incidents. The author himself asked whether such metric is relevant to indicate the quality of the organization security.

In [13], Bernsmed *et al.* propose a security SLA for federated Clouds. Their approach is based on security mechanisms divided into three resource types *i.e.,* Storage, Processing and Network. For instance, one network security mechanism is integrity protection but the objective is stated in textual form *e.g., All text messages will be digitally signed.*

To our knowledge, security service requirements works are aligned on standards and recommendations (*e.g.,* Cloud Security Alliance, NIST, Common Criteria) where security is expressed in natural language. These work can be viewed as a list of good practices which may relatively improve the security but cannot enforce end-to-end security with strong guarantees (*e.g.,* with formal proof).

Security Policy Standards

Specific standards have been elaborated to express security policies for distributed web services.

WS-Security [2] aims at providing *quality of protection* to SOAP (Simple Object Access Protocol) messages[1]. The WS-Security standard describes for example how to encrypt a message (for confidentiality), how to sign a message (for integrity) or how to

[1] SOAP is a protocol based on XML for exchanging structured information.

attach security tokens to a message. Therefore, it contains a low-level specification of how the security is implemented (*e.g.,* which cryptographic algorithm, what size of keys) and do not focus on the security objectives to achieve.

XACML [89] (for eXtensible Access Control Markup Language) is an OASIS standard defining access control policies in XML. It comprises access rules, roles and delegation concepts. One goal is to promote interoperability between access control implementations by multiple vendors. In XACML, a rule may be conditional to be triggered in reaction to a situation or event. The functionalities are similar to Ponder [39]. Both XACML and Ponder focus on access control which is not sufficient to enforce end-to-end security objectives (*i.e.,* information flows).

Model-Driven Security

For a decade, model-driven security has been investigated as a mean to help engineers integrate security concerns at design. SecureUML [8, 82] is designed to specify authorizations policies based on RBAC extended with constraints. Constraints are expressed in OCL (Object Constraint Language) and mapped to a first-order logic to be able to use formal verification such as theorem-proving tools. This is a good step toward automatic security. But, SecureUML is limited to expressing access controls for applications. It cannot help secure with existing applications and cannot tackle high-level objectives such as information flows.

Interestingly, Nguyen *et al.* have conducted a system review of model-driven security in [92] by selecting the 80 most relevant papers from 10.633 relevant papers. They found that:

- Most papers focus on authorization and confidentiality while only a few address other security concerns (*e.g.,* integrity, availability, authentication).
- Most security modeling languages lack a thorough semantic foundation (which the authors recognized to be important).
- An integrated approach for the generation of functional code and security infrastructures was incomplete.

According to the authors, the literature is limited to specific, isolated security concerns, and lack formality, automation, process-integration and evaluation.

Most of the literature focuses on designing secure applications whereas our goal is to design secure systems composed of applications. To our knowledge, no model-driven work has tried to tackle larger systems with information flows as security objectives.

2.1.2 Component-based Models

Many frameworks and models have been proposed to design applications in virtualized distributed systems.

A popular approach is to use **component-based models** [113]. A **component** is a functional piece of code exposing its interfaces to communicate or coordinate with other components. The idea is to have *reusable* modules.

One approach is to use components to automatically deploy and configure the application stack within virtual machines. These tasks are traditionally done using hand-made scripts. For example, in [45], Etchevers *et al.* presents VAMP, a Virtual Applications Management Platform. Automated steps are the generation of VM images[2], launch of VM instances[3], local configuration of software and global configuration of dependencies between components. The OASIS standardization consortium has defined a Topology and Orchestration Specification for Cloud Applications (TOSCA)[4] [18] to enhance the portability and management of Cloud applications and services across Clouds and during their whole lifecycle. Many works are based on TOSCA to propose a runtime for TOSCA-based Cloud applications [17] or provide a modeling tool to ease the specification [72]. In [119], Waizenegger *et al.* define security policies in TOSCA and propose a mechanism for automatic processing of these policies. As an example, the authors describe a region policy constraining VMs to be instantiated in specific geographical regions (*e.g.,* EU, US). We do not consider geographical constraint as security. It does not provide any security feature and cannot fit alongside properties like confidentiality or integrity. Instead, we consider it as an environmental constraint such as energy power. Similar to TOSCA is MODAClouds [38] which supports the design of multi-cloud applications and aims at automatic provisioning, deployment and adaptation at runtime on the basis of an application model.

Component-based models are sound and appealing. Many complex frameworks have been developed. However, as our primary concern is security, we prefer to focus first on designing a much simple but security-centric model rather than investing extensive engineering effort to adapt existing frameworks for security. Integrating our findings back to standard frameworks is part of our future work though.

As model-driven security before, component-based models follow a model-driven approach we detail thereafter.

2.1.3 Model-Driven Engineering and Metamodeling

Thanks to object-oriented programming, engineers and computer scientists are familiar with the notions of abstraction and representation. They define *classes* or *objects* and instantiate them *i.e.,* each attribute has a value. For example, one can define vm_{alice} and vm_{bob}: instances of a class **VM** where attribute *vcpu* is set to 2 and 4 respectively. A **metamodel** is a *model of models* and a model is designed to be an *instance of* a *metamodel.* In the previous example, **VM** is a model of vm_{alice}, vm_{bob} and the object **Class** is a model of **VM**. Indeed, we can define other classes *e.g.,* **Network**, **Data** instances of **Class**.

As depicted in Figure 2.1, metamodeling is divided into 4 levels (from M0 to M3) with:

- M0 – System. It is the **real** system without simplification or abstraction. In our context, M0 includes the real (physical) infrastructure, the virtual machines (as implemented in KVM or Xen) and virtual networks.

[2] A VM image is a virtual copy of a physical hard disk drive. It contains complete contents and data structure of a hard disk.

[3] An instance is a running VM obtained by executing a VM image.

[4] `https://www.oasis-open.org/committees/tosca/`

- M1 – Model. It is a simplified representation of the reality. As presented above, the representation of two VMs (called vm_{alice}, vm_{bob}) is a model.
- M2 – Metamodel. It is an abstraction of models. Using the previous example, the object VM is a metamodel of the model (vm_{alice}, vm_{bob}).
- M3 – Metametamodel. Also called Meta-Object Facility (MOF), it is a model to describe metamodels. For example, UML (Unified Modeling Language) allows us to define the object VM and thus, it falls into the metametamodel category. There is no need for upper levels as metametamodels can represent themselves *e.g.,* UML can be modeled in UML.

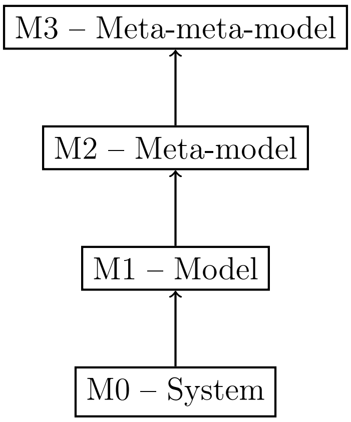


Figure 2.1: Metamodeling hierarchy

We have previously said that component-based models and model-driven security follow a model-driven engineering methodology. **Model-Driven Engineering** (MDE) is a software development methodology that aims at automating complex programming tasks such as providing support for system persistence, inter-operability, and distribution [3]. Its foundation is the metamodeling paradigm. Reducing complex programming tasks can be achieved by abstracting system-specific constraints and providing automatic transformation. For example, an engineer often faces the problem of designing a multi-platform application. In a model-driven approach, he will design a generic model of the application and use automatic tools to refine it into system-specific versions.

2.1.4 Our Unified Metamodel

Security specification is often complex and requires field expertise. For instance, SELinux exists for decades and is recognized as a great security mechanism. But writing a correct SELinux policy is complicated even for a knowledgeable administrator. We believe that if a security specification tool is simple to use, it will be an incentive to a wider adoption. Therefore, we acknowledge this challenge and work toward a more user-friendly interface.

To do so, we follow a MDE approach to have a unified view of a security-aware system and automatically process it to obtain a secure deployment. In addition, we benefit from MDE to generate a multi-platform (Windows, Linux and MacOS) modeling tool with automatic upgrades as metamodels are progressively enriched with new constraints or features. For this purpose, we use the **Eclipse Modeling Framework** (EMF) to generate and maintain our modeling toolbox called **Sam4C** (Security-Aware Models for Clouds)[5] [79].

Sam4C is designed to be easy to pick up for end users like the Seed4C partners hence it offers a graphical interface to model the application and a textual language to specify the security requirements. The graphical interface is generated using **GMF**[6] (Graphical Modeling Framework) and the textual language (called Domain Specific Language) is generated using **Xtext**[7]. Both frameworks are based on EMF as illustrated in Figure 2.2. An EMF *resource* is a persistent document containing modeled contents. It is *abstract*: it is only a template (*i.e.,* list of classes and methods). We depicted two existing implementations: *XMIResource* and *XtextResource* which generate respectively an *XMI* file (XML Metadata Interchange) and a *text* file. It implies that the same model may be exported in XMI (this is the default implementation) or text (providing a grammar). Finally, the model can be manipulated graphically with a *GMF Editor.*

The underlying metamodel is described in UML and stored in an *ecore* file. The security part is presented as a DSL (Domain Specific Language) using Xtext whereas the application part is presented graphically using GMF. The important point is that every element (textual, graphical or none of the two) are specified in single unified UML model: our metamodel.

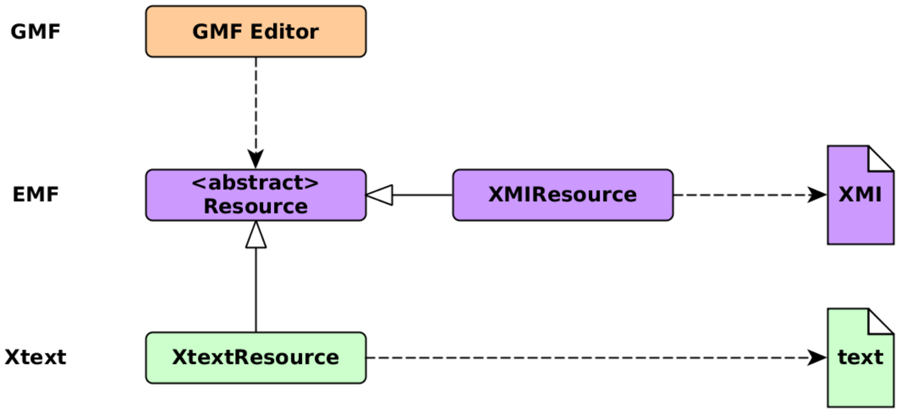


Figure 2.2: Integration of GMF and Xtext with EMF

To begin with our presentation, Figure 2.3 is part of the UML metamodel for security-aware virtualized applications (without the infrastructure). In a MDE approach, all objects must be contained in a root object. In Sam4C, **All** is the root object containing our two-fold description:

- The architecture part describing the components of an application and their relations
- The security part describing the components of security requirements and their relations.

[5] Demo Videos at `https://www.youtube.com/playlist?list=PLXdZxOWBaqWpJCRs_OpeBeHOWtba4yHSq`

[6] `https://wiki.eclipse.org/GMF`

[7] `https://eclipse.org/Xtext/`

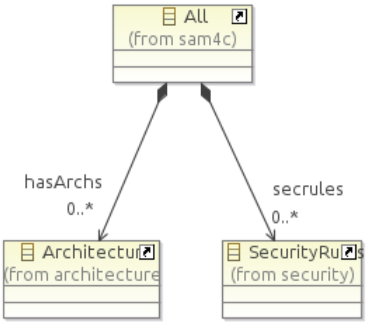


Figure 2.3: Root objects of the security-aware virtual distributed system metamodel in UML

In the following of this chapter, we present our metamodels of virtualized distributed systems (virtualized application and infrastructure) in Section 2.2 and their security in Section 2.3.

2.2 Modelization of Virtualized Distributed Systems

In this section, we present our metamodels of a **virtualized distributed system** which is composed of the tenant's **virtualized application** and the **virtualization-based infrastructure** it is deployed on.

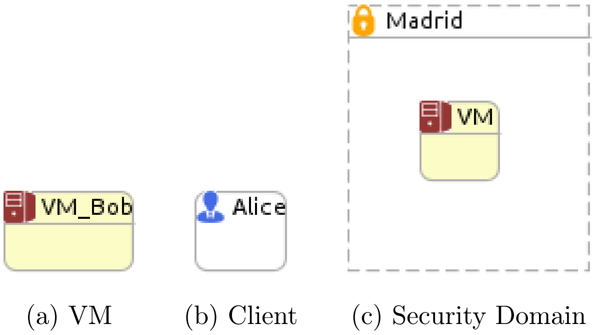
2.2.1 Virtualized Application Metamodel

Our virtualized application metamodel is composed of two layers: the *virtual layer* and the *application layer*. Our metamodel must be able to represent a n-tier virtualized application.

Virtual Layer

At the virtual layer, the modeling consists in describing a set of interconnected virtual machines. Our model contains several node types:

- **VM** (Figure 2.4a). It encapsulates the application data and processes. As depicted in Figure 2.6, it is characterized by:
  - *image*. This is the disk image from which the VM is instantiated.
  - *vcpus*. The processing power requirement is usually specified as a number of virtual cpus in Clouds.
  - *ram*. The VM requires memory space to run.
  - *storage*. This is the disk space attached to the VM.
  - *location*. A VM may be constrained to be deployed only in a specified region (*e.g.,* France, USA) for legal purpose.



(a) VM (b) Client (c) Security Domain

Figure 2.4: Virtual Layer Node Components

- **Client** (Figure 2.4b). A virtualized application must usually be accessible remotely. This entity represents an external endpoint and how the network should be configured accordingly.
- **Security Domain** (Figure 2.4c). It is an abstract container for virtual machines and networks. It is useful to specify security properties on a group rather than on each individual entity. For example, one may want to specify that part of his application is isolated from the other.

The last entities of our virtual layer are networking components. As depicted in Figure 2.5, instead of representing a graph of VMs, we explicitly model the network, called **VNet**, as a circle (INTRANET on the figure). A VM has (one or multiple) virtual interfaces called **Veth** (ETH0 on the figure) where each virtual interface is linked to a single virtual network via a **VLink**. Such modelization allows modeling two VMs to communicate through two different virtual networks *e.g.,* a “public” network and an administration network, with distinct security properties.

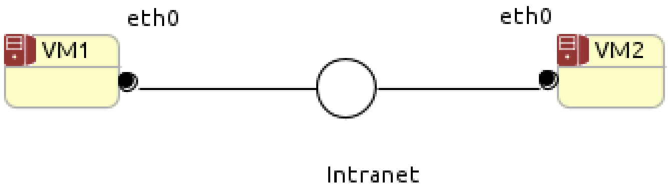


Figure 2.5: Virtual Network Components

Figure 2.6 is our virtual layer metamodel in UML. Any entity (VM, VNet, etc.) inherits from the abstract object **Element**, they all have a *name*, and composes the *architecture* root node.

Application Layer

At the application layer, the modeling consists in describing **Data** (*e.g.,* files, databases) and **Services** (*e.g.,* processes such as database engines) composing the application at a

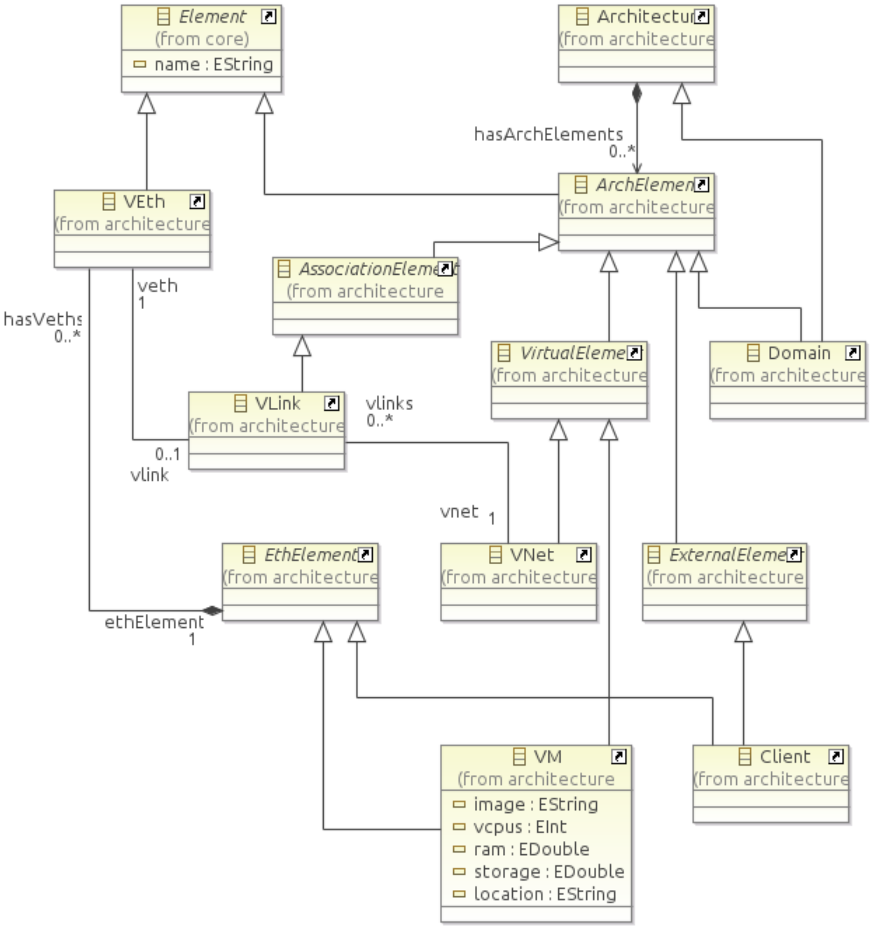


Figure 2.6: Virtual Layer Metamodel

granularity relevant to security. Application entities may be logically grouped into an **AppDomain** or linked with an **AppLink** to model the interaction between entities. Figure 2.7 depicts an AppDomain called System with an SSH service reading or writing Logs as represented by the green AppLink.

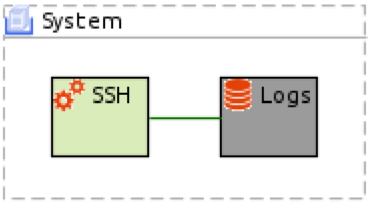


Figure 2.7: System application domain with SSH service accessing Logs data

Figure 2.8 is our application layer metamodel in UML. Similarly to virtual entities, any application entity (Data, Service, etc.) inherits from the abstract object **Element**, they all have a *name*, and composes either the *architecture* (for describing applications only) or the **VM**.

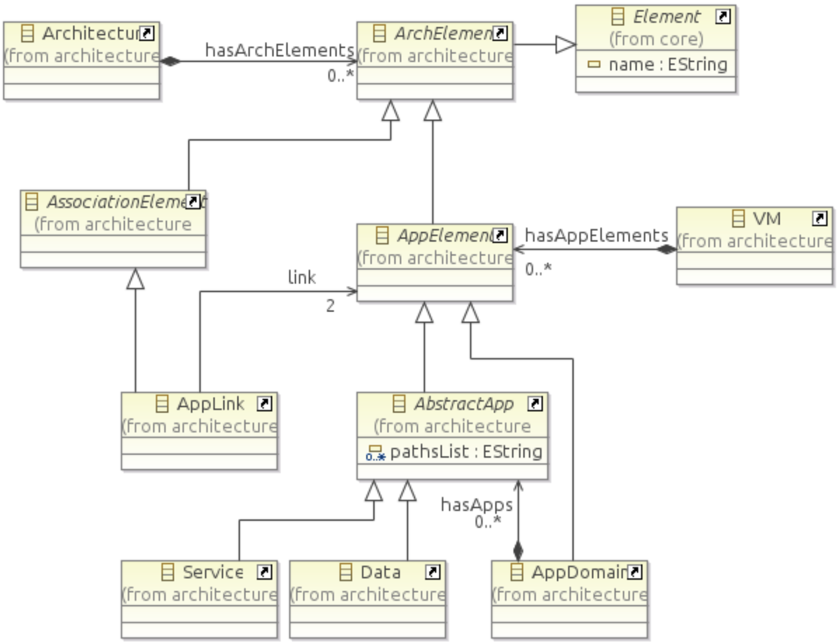


Figure 2.8: Application Layer Metamodel

2.2.2 Infrastructure Metamodel

Our virtualization-based infrastructure metamodel describes the platform virtualized applications are deployed on. It is divided into two layers, the *infrastructure layer* and the *microarchitecture layer*.

Infrastructure layer

Figure 2.9 is the root part of the infrastructure metamodel. Our model comprises:

- **Node**. Also referred as host or physical machine, it is characterized by *disk* space and a *location*. The location is not sensu stricto a security concept, and thus not included in the security part. However, due to differences in region or country legislation, it is important for users to specify a location constraint.

- **INet** (Infrastructure Network). In a virtualized infrastructure, virtual networks such as Software Defined Networks (SDN) [53] can be instantiated between virtual machines. Multiple VNets may be mapped to the same INet. We discuss in Section 4.4 a preliminary solution where any constraint-free VNet with an external endpoint (*e.g.*, a client) may be mapped on the same *public* INet.

```
<?xml version="1.0" encoding="UTF-8"?>
<infra:Infrastructure xmi:version="2.0" xmlns:xmi="http://www.omg.org/
   XMI" xmlns:infra="http://avalon.inria.fr/infra/">
  <hasNodes name="taurus-0" disk="598.0" location="lyon">
  </hasNodes>
  <hasNodes name="taurus-1" disk="598.0" location="lyon">
  </hasNodes>
  <hasINets name="Public"/>
```

```
</infra:Infrastructure>
```

Listing 2.1: Infrastructure XML with 2 Nodes and 1 INet

The infrastructure metamodel has no graphical view as it is not exposed to the client and can be automatically generated. Listing 2.1 is the XML representation of an infrastructure composed of 2 Nodes (*i.e.,* taurus-0 and taurus-1) located in Lyon with 598Gb disk storage each and a single public INet.

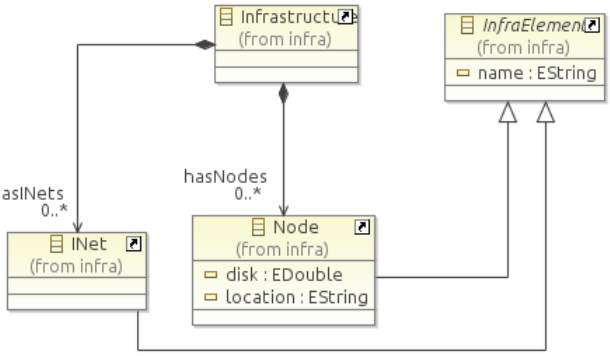


Figure 2.9: Infrastructure Layer Metamodel

Microarchitecture layer

In our approach, one way of satisfying isolation security properties is by placement. A class of security attacks consists in exploiting the shared microarchitecture to conduct cross-VM exploits (more details in Section 4.2). Traditional approaches from the literature ensure isolation through collocation/anti-collocation constraints [28, 64]. These approaches never consider a more fine-grained allocation *i.e.,* exploiting the *microarchitecture* to improve the consolidation and tackle the very reason for cross-VM attacks.

The **microarchitecture** is the internal organization of a physical machine. It includes the implementation of the processor and peripheral devices (*e.g.,* I/O devices).

Nowadays, most physical machines are *multiprocessors i.e.,* computers with two or more processing units sharing main memory and peripherals, in order to simultaneously process programs. Our model comprises both the two existing shared memory access models: SMP (Symetric MultiProcessing) and NUMA (Non-Uniform Memory Access).

A **SMP architecture** possesses multiples parallel cores connected to a *single* main memory (usually accessed uniformly across cores) through a memory bus and multiple levels of cache in between as illustrated in Figure 2.10.

A **NUMA architecture** is a logical evolution of the SMP architecture where shared memory is decomposed in multiple sockets and memory access latencies differ when a core accesses a local NUMA node or a remote NUMA node. As depicted in Figure 2.11, access to distant memory is done through an interconnection network.

For instance, Figure 2.12 is a model of a traditional SMP architecture as obtained from the Hwloc tool[8] [24]. There is a single memory of 8005MB and 2 processor sockets

[8] http://www.open-mpi.org/projects/hwloc/

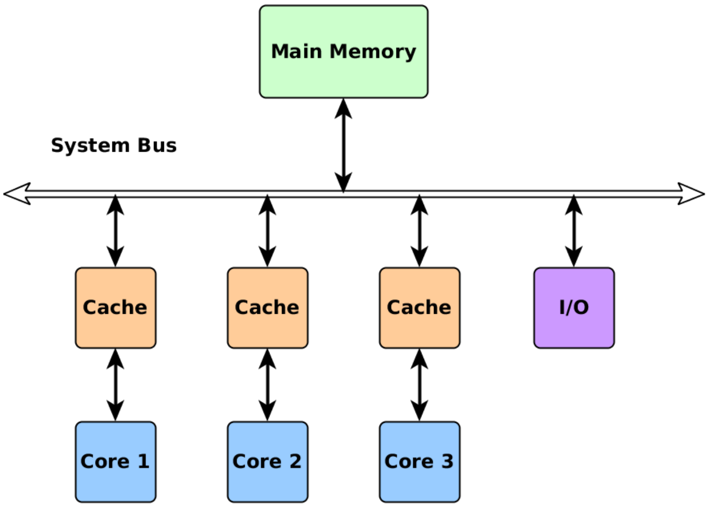


Figure 2.10: SMP architecture principles

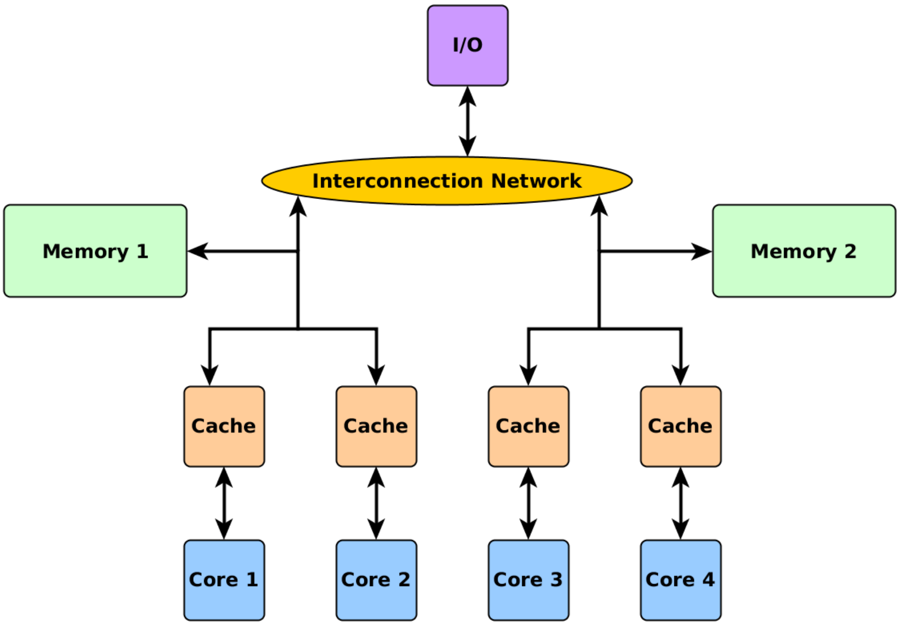


Figure 2.11: NUMA architecture principles

each one composed of 4 cores with a private level 1 (L1) cache of 32KB per core and a level 2 (L2) cache of 6144KB per pair of cores. The element **PU** is a processing unit. It can be seen as a thread inside a core. In the architectures we address, there is one or two PUs. This element is irrelevant to our model as we do not take them into account for our placement decision. Indeed, as explained in [97], nearly all providers disable the possibility of sharing PUs due to the easiness of building a L1 covert channel.

Figure 2.13 is a NUMA architecture with two NUMA nodes of 16GB each (total of 32GB). Each processor socket has 6 cores sharing a level 3 (L3) cache of 15MB with private L1 (32KB) and L2 (256KB) caches per core.

To represent both architecture types, our microarchitecture metamodel depicted in Figure 2.14 explicitly defines two types of micro-elements, **Core** and **Numa**, as part of

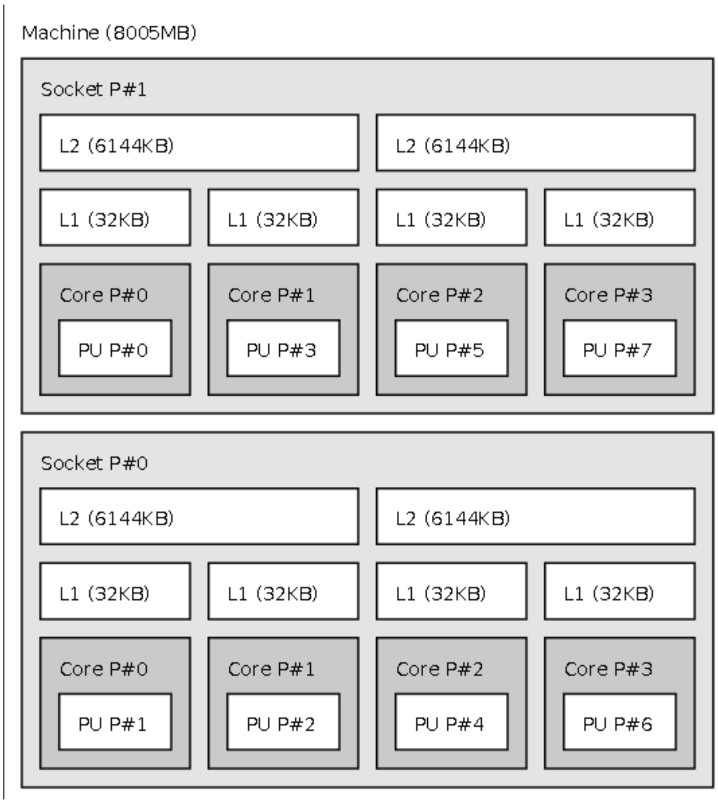


Figure 2.12: Intel Xeon E5420 QC Uniform Memory Access topology (Grid'5000 Genepi Node) from Hwloc

a **Node**. We view the SMP architecture as a NUMA architecture with a single memory. The placement of a VM onto a host is realized by selecting a configuration of NUMAs and Cores that will be allocated to the VM.

As an example is Listing 2.2, the extension of the infrastructure model presented in Listing 2.1 with microarchitectural elements (*i.e.*, cores and NUMAs). As the reader may notice, *Taurus* nodes are the simplified representation of the Intel Xeon E5-2630 presented in Figure 2.13.

```
<?xml version="1.0" encoding="UTF-8"?>
<infra:Infrastructure xmi:version="2.0" xmlns:xmi="http://www.omg.org/
   XMI" xmlns:infra="http://avalon.inria.fr/infra/">
  <hasNodes name="taurus-0" disk="598.0" location="lyon" availableNumas
     ="//@hasNodes.0/@hasNumas.0 //@hasNodes.0/@hasNumas.1"
     availableCores="//@hasNodes.0/@hasCores.0 //@hasNodes.0/@hasCores
     .1 //@hasNodes.0/@hasCores.2 //@hasNodes.0/@hasCores.3 //@hasNodes
     .0/@hasCores.4 //@hasNodes.0/@hasCores.5 //@hasNodes.0/@hasCores.6
      //@hasNodes.0/@hasCores.7 //@hasNodes.0/@hasCores.8 //@hasNodes
     .0/@hasCores.9 //@hasNodes.0/@hasCores.10 //@hasNodes.0/@hasCores
     .11">
    <hasNumas name="0" memtotal="16384.0" memfree="16384.0"/>
    <hasNumas name="1" memtotal="16384.0" memfree="16384.0"/>
    <hasCores name="0"/>
    <hasCores name="1"/>
    <hasCores name="2"/>
    <hasCores name="3"/>
```

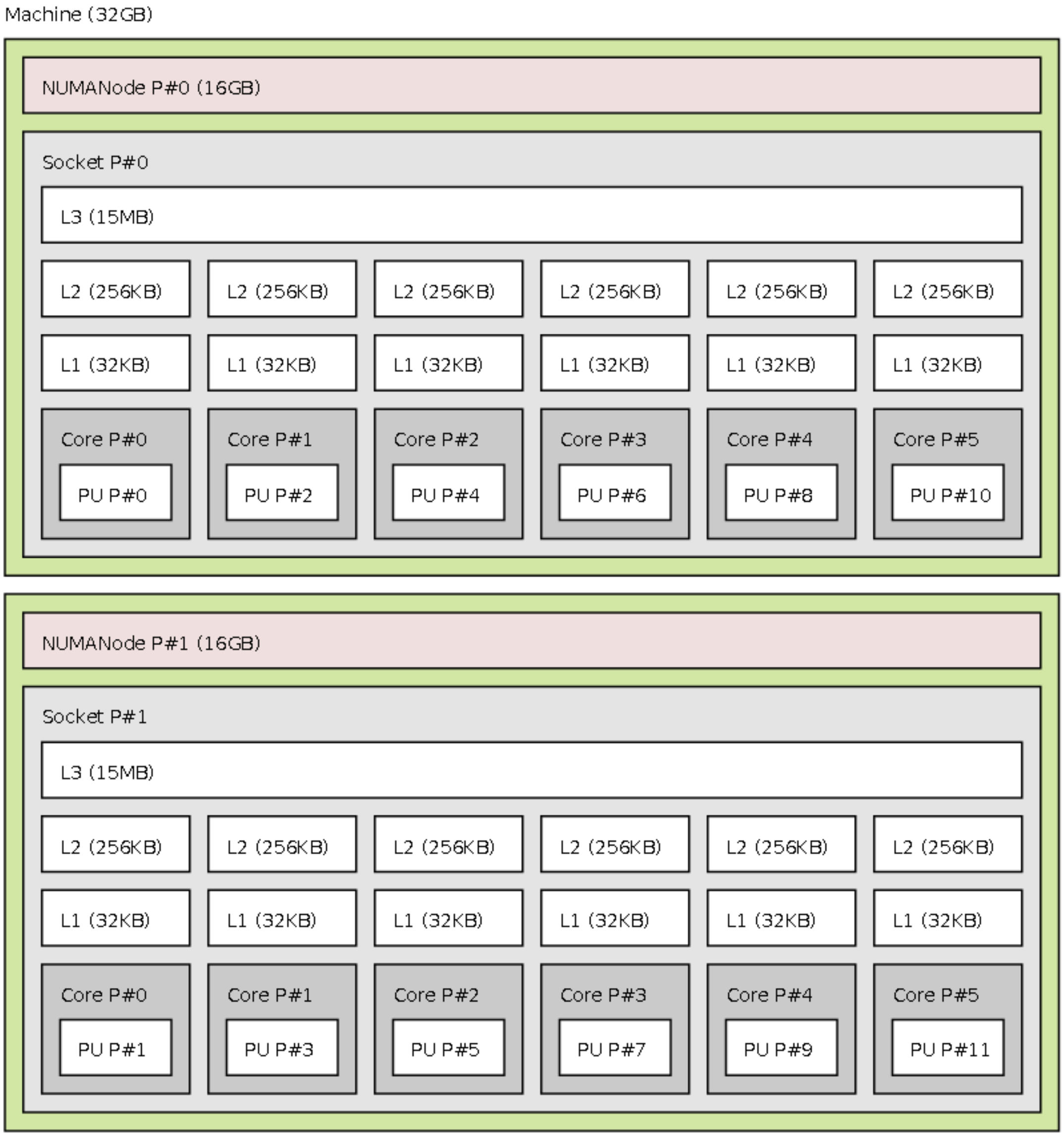


Figure 2.13: Intel Xeon E5-2630 Non-Uniform Memory Access topology (Grid'5000 Taurus Node) from Hwloc

```
    <hasCores name="4"/>
    <hasCores name="5"/>
    <hasCores name="6"/>
    <hasCores name="7"/>
    <hasCores name="8"/>
    <hasCores name="9"/>
    <hasCores name="10"/>
    <hasCores name="11"/>
  </hasNodes>
  <hasNodes name="taurus-1" disk="598.0" location="lyon" availableNumas
     ="//@hasNodes.1/@hasNumas.0 //@hasNodes.1/@hasNumas.1"
     availableCores="//@hasNodes.1/@hasCores.0 //@hasNodes.1/@hasCores
     .1 //@hasNodes.1/@hasCores.2 //@hasNodes.1/@hasCores.3 //@hasNodes
     .1/@hasCores.4 //@hasNodes.1/@hasCores.5 //@hasNodes.1/@hasCores.6
```

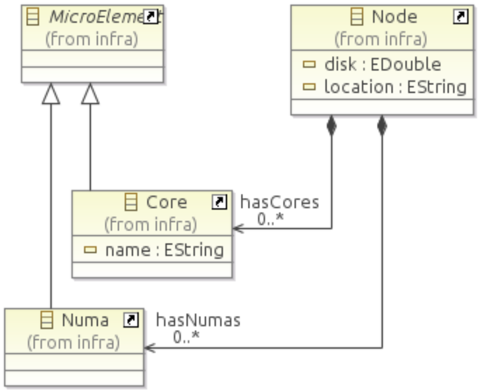


Figure 2.14: Microarchitecture Metamodel

```
      //@hasNodes.1/@hasCores.7 //@hasNodes.1/@hasCores.8 //@hasNodes
     .1/@hasCores.9 //@hasNodes.1/@hasCores.10 //@hasNodes.1/@hasCores
     .11">
    <hasNumas name="0" memtotal="16384.0" memfree="16384.0"/>
    <hasNumas name="1" memtotal="16384.0" memfree="16384.0"/>
    <hasCores name="0"/>
    <hasCores name="1"/>
    <hasCores name="2"/>
    <hasCores name="3"/>
    <hasCores name="4"/>
    <hasCores name="5"/>
    <hasCores name="6"/>
    <hasCores name="7"/>
    <hasCores name="8"/>
    <hasCores name="9"/>
    <hasCores name="10"/>
    <hasCores name="11"/>
  </hasNodes>
</infra:Infrastructure>
```

Listing 2.2: Infrastructure XML with 2 Nodes and 1 INet

2.3 Modelization of Security Requirements

There is a vast range of security mechanisms which have been proven efficient in practice. Each mechanism can guarantee specific security properties (*e.g.,* confidentiality, integrity) for particular entities. For example, cryptography may protect data in transit or at rest against unauthorized accesses but cannot be employed for controlling applications' behavior. We believe a key concept is to deliver *on-demand* security: a user should be able to specify what is the required secure behavior of his system. To do so, we rely on existing security mechanisms to enforce security properties. The process of specifying/configuring a security policy is usually very complex and error-prone. Even with security expertise, a human can still realize unintended misconfiguration leading to security breaches. For this very reason, security policies should be enforced in an automatic fashion *i.e.,* , where human intervention only occurs if strictly necessary. Though the intended behavior of a system *i.e.,* what a system should or should not do, falls into human knowledge.

In this Thesis, we follow a *specification-driven* approach: the user specifies the security he wants without knowledge of how it will be enforced. Halpern *et al.* [57] state that security policies described in a natural language have quite ambiguous semantics. On the other hand, a formal language or logic can provide clear syntax and semantics. Therefore, our specification language should be logic-based.

In this section, we review existing models and languages to specify security properties and discuss important security properties. Then, we present our security metamodel: how to uniformly address different security mechanisms to enforce the same property. Finally, we define precisely *what does mean* a security property by proposing a new logic-based formalization.

In our *specification-driven* approach, the user describes components of a *virtualized application* and its security. Similar to our system-agnostic modelization of applications, we propose an *enforcement-agnostic* modelization of security requirements. As depicted in Figure 2.15, though the enforcement process itself is driven by the security policy, it applies to the real system. Therefore, we must link entities described in the model to their real implementation counterpart. To do so, we propose a domain-specific language (DSL) to specify security policies.

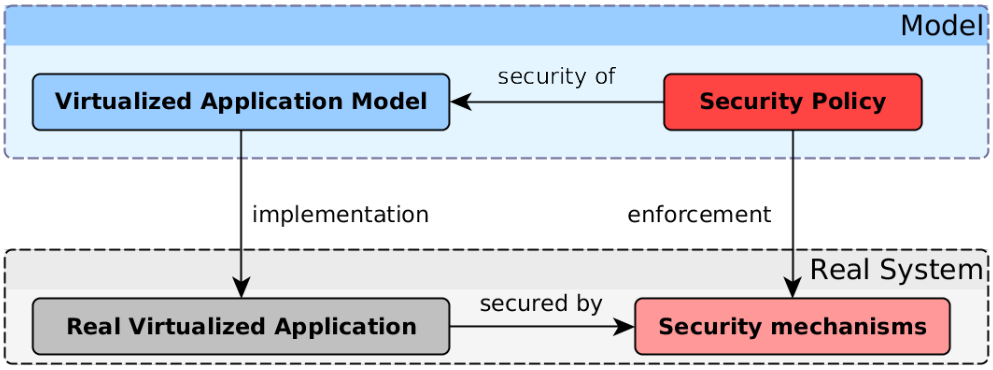


Figure 2.15: Model-to-system workflow

2.3.1 Attribute-based Contexts

Our security DSL must address any entities/resources independently from system-specific resources naming to address any systems. For example, one can describe a **Service** entity called SSH without specifying what SSH is or where it is *physically* located. In most Linux architectures, the SSH service daemon is located in `/usr/sbin/sshd` and the configuration files are in `/etc/ssh`. For security specification, we want the real path to be abstracted. Our solution is to use **attribute-based contexts**.

A **context** is an identifier referring to a (single or) group of entities. Security properties of our language apply to contexts. Therefore, entities with the same context are constrained by the same properties. An **attribute-based context** is composed of key-value pair of attributes where each attribute provides information about the resources it identifies. In a *virtualized application* model, the user can specify components of an application using *names*. For instance, Figure 2.16 is the model of a service named SSH contained in an application domain named SYSTEM and packed into a virtual machine named VM1. Then, the context identifying the SSH service is `(VM="VM1"):(AppDomain="System"):(Service="SSH")`.

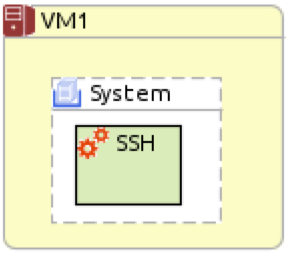


Figure 2.16: Model of a SSH service in a System application domain inside VM1 virtual machine

Any node elements of our *virtualized application* metamodel is an attribute: **Domain, VM, Client, VNet, Veth, AppDomain, Data and Service**. This metamodel contains only static architectural elements and it is often useful to refer to other characteristics. Typically, a system user has different rights whether he is a standard user or an administrator. Therefore, our language allows defining new attributes as follows:

```
#attribute Role = (StandardUser, SystemAdmin, SysnetAdmin);
```

In the above example, a new attribute `Role` has been defined and it accepts 3 values `StandardUser, SystemAdmin, SysnetAdmin`.

Using attributes, new contexts can be defined. Here is an example with a context `ctxAdmins` addressing any entity with the `SystemAdmin` role:

```
#context ctxAdmins = (Role="SystemAdmin");
```

Any context may be reused to form new, more precise contexts. Here is an example with `ctxSSH_Admin` referring to the SSH service (of System of VM1) with the `SystemAdmin` role:

```
#context ctxSSH_Admin = (VM="VM1"):(AppDomain="System"):(Service="
   SSH"):ctxAdmins;
```

Attributes values are regular expression strings. In the previous scenario Figure 2.16, instead of specifying SSH of VM1, any VMs' SSH service may be selected using the *star* character: `(VM="*"):(AppDomain="System"):(Service="SSH")`.

To uniquely address an element defined in a *virtualized application* model, a context is derived from the model's hierarchy. In our example, SSH derived context is `VM1.System.SSH`. It follows the same grammar as other contexts, for example:

```
#context ctxSSH_Admin = VM1.System.SSH:ctxAdmins;
```

To sum up, Figure 2.17 presents the metamodel security part limited to context and attribute definition. In our metamodel, a **Context** is a list of either:

- **References** *i.e.,* contexts' names defined with the `#context` keyword.

- or **ContainmentReferences**. It can be a **ValuedAttribute** (*e.g.,* `(AttributeKey=Value)`) or an **ElementList** which is a list of **Elements** from the virtualized application metamodel (*e.g.,* VM, Domain, Data, Service).

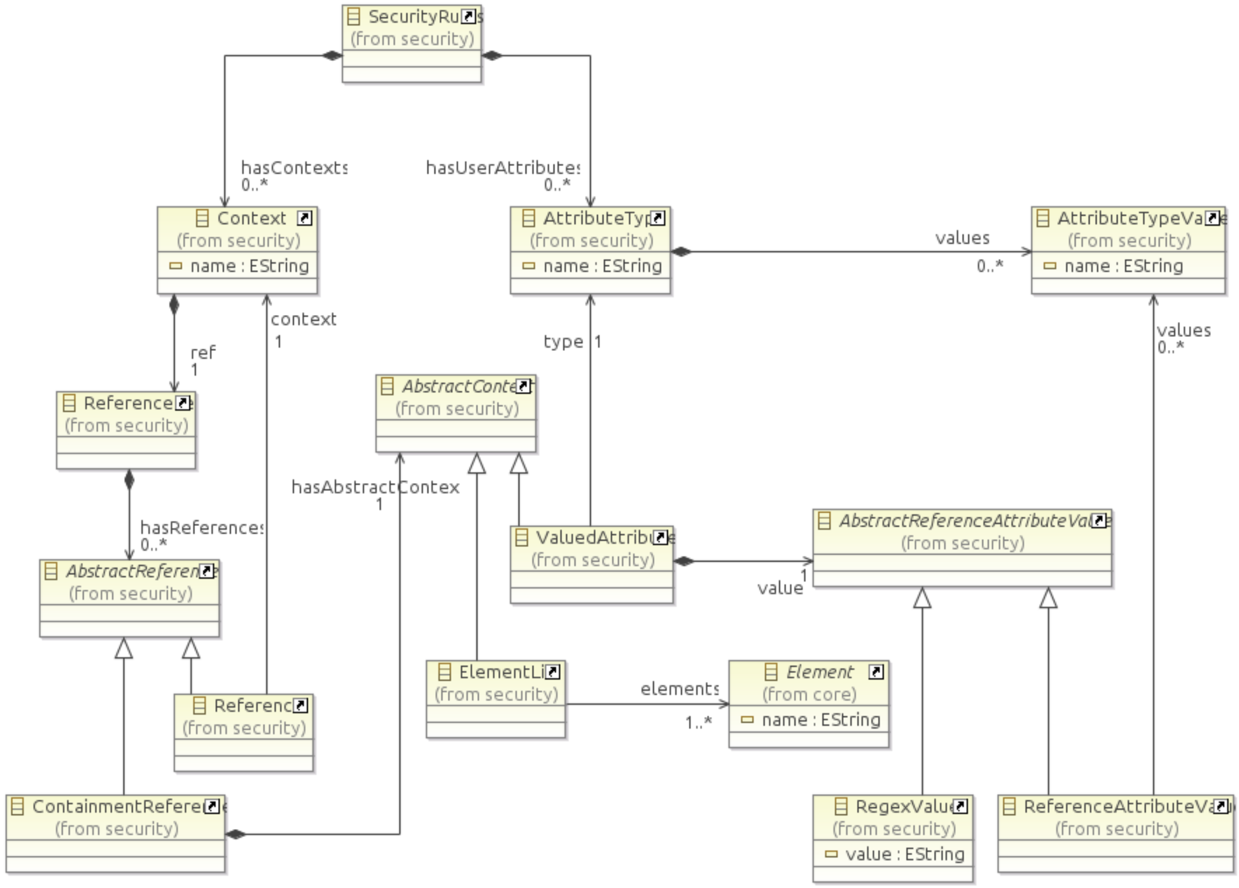


Figure 2.17: Security attributes and contexts metamodel

Bindings

A context represents a set of entities with common characteristics. We call **binding** the association of a context to "real" entities. For instance, the *binding* of `ctxSSH_Admin` could be:

```
/usr/sbin/sshd ctxSSH_Admin
/home/*/.ssh/* ctxSSH_Admin
```

In our metamodel, the *binding* relation is depicted in Figure 2.18 where a binding is a path and multiple bindings can be associated to the same context.

2.3.2 Security Properties

In our language, a **security property** is a property on *contexts* we model as a security function with parameters (*i.e.,* contexts). Semantics of properties are explained after in

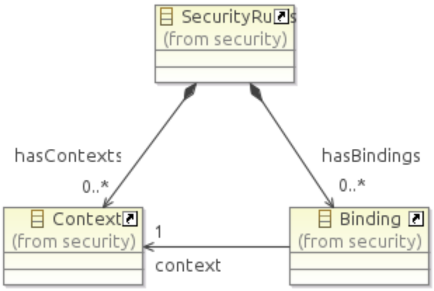


Figure 2.18: Binding metamodel

Chapter 3, we use informal definitions in this section.

Confidentiality, Integrity and Isolation The common characteristics of these 3 properties are *what is secure* and *what is authorized*. Their template is `Property(ctxSecured, ctxAuthorized)`. For example, the confidentiality property protects against unauthorized read accesses. Suppose only a user with ADMIN role is allowed to read a LOG file. In our language, the property is:

```
#property Confidentiality((Data="Log"), (Role="Admin"));
```

Authentication This property has the particularity of *changing an entity's context*. Typically, a user can log as an admin hence changing his role from standard user to admin. This authentication is realized by a third entity. The authentication property has 3 parameters that are *what is the source context*, *what is the authentication process* and *what are the valid destination contexts*. Suppose an unknown user wants to authenticate using SSH and acquire system-user or admin rights, then the corresponding property is:

```
#property Authentication((Username=""), (Service="SSH"), (Role="
   Admin|SystemUser"));
```

Model-based Authorization Inference One of our goals is to ease the specification for the user. The virtualized application model provides useful operational information which could appear redundant in the security policy. For example, Figure 2.19 depicts an application with $VM1$ connected to a $Client$ and $VM2$ through respectively $Internet$ VNet and $Intranet$ VNet. It is natural to assume that a security policy should allow $VM1$ to access $Internet$ and $Intranet$: this information from the model is *implicit*.

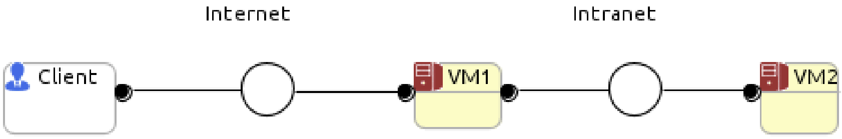


Figure 2.19: Virtualized Application Model with 2 VNets, 2 VMs and 1 client.

Suppose we want $VM1$ to be isolated from any network, the implicit exceptions ($Internet$ and $Intranet$) can be deduced from the model. Therefore, we introduce the concept of *implicit properties* with only one parameter *i.e.,* what is secure; authorizations are deduced from the virtualized application model. Hence, the following implicit property can be written:

```
#property Isolation(VM1);
```

Then, an automatic transformation procedure (presented in Section 4.1)infers the list of exceptions (*i.e.,* authorized elements) from the virtualized application model to produce an equivalent *explicit* property:

```
#property Isolation(VM1, {Internet, Intranet});
```

This transformation is only viable for simple properties (*i.e.,* Confidentiality, Integrity, Isolation) without user-defined attributes (*e.g.,* Role).

Properties with Grades As explained after in Chapter 3, our security properties have a unique and clear interpretation. Intuitively, the enforcement of a security property should guarantee a secure system. However, in practice, there is no perfect security but rather a scale from weak to strong security. Indeed, nearly all systems rendering a service can be exploited providing sufficient time and resources. For example, cryptography is used to guarantee Confidentiality and Integrity of data. The quality of protection differs depending on the cryptographic algorithm used (*e.g.,* AES, RSA) and the size of the key(s). It is said to be computationally secure *i.e.,* it is theoretically possible but either it would take too much time or there is no known solutions yet. Even if a protection mechanism is considered to be foolproof, humans using the system are far from it. This quality of protection is encompassed in security practices by the notion of risk. Many risk-evaluation frameworks have been proposed for virtualized environments [109, 130]. The common principle is to associate the likelihood of an asset to be attacked with a security counter-measure of equal or superior strength thus diminishing the risk by making the attack "not worth it". The vast range of existing risk metrics highlights the difficulty of defining a good and sound one. The main reason for evaluating risks is to deploy cost-effective security measures. For an organization, security is often disregarded due to its cost and its absence of visible benefits.

In this Thesis, we do not want to improve generic risk metrics and evaluations State of the Art. Yet, as we tackle Cloud-like platforms, we must consider the financial impact of security and thus allow a user to specify his security requirements with different quality of protection. Accordingly, we propose to simply extend our properties with a **grade**. The grade is specified by the end-user hence it should be independent from the type of the protected entities or the enforcement mechanism. We believe a coarse-grain grading system (*e.g.,* Low, Medium, Strong) is more relevant and offers a good compromise. For future flexibility, we implement a grade as an integer value. Informally, a higher grade means a stronger security and a lower grade means a weaker security. Then, our grades can be specified as follows:

```
#property Isolation(VM1, grade);
```

Properties without explicitly specified grades are implicitly associated with the highest value (*e.g.,* 100). In preliminary work [29], we have first envisioned the grade as a vector of security mechanisms but this approach has been discarded in this Thesis as it is not mechanism-agnostic. However, in Chapter 4, we propose a metric for a specific type of attacks called covert-channels and we discuss its relation to our grading system. Nevertheless, the problem of defining more relevant grades or generic risk metrics is part of our future work.

2.4 Conclusion

We have shown the necessity of a specification-driven approach where a user expresses its security constraints in a simple and clear way. These constraints are high-level security objectives, agnostic of any enforcement mechanisms, that apply to elements of an application which needs to be modeled as well. In the literature, there is no security aware application models suitable to enforce the end-to-end security of virtualized applications. Instead, the literate only exhibits application models without security or security models without application descriptions. Accordingly, we have proposed a unified metamodel embracing both the virtualized application and its security policy.

Furthermore, we have detailed the infrastructure metamodel hosting virtualized applications. This metamodel is used and extended in Chapter 4 to integrate security solutions. In particular, we will show how to enforce security properties between VMs with a security-aware placement algorithm and how to configure security mechanisms according to the security policy.

But before, as the main drawback of the literature is the lack of formalism, we propose in the next chapter a logic giving a unique and clear description of what confidentiality, integrity or isolation means. Then, this logic is used in Chapter 4 to refine our global security policy into multiple local properties enforceable by placement or by configuring local security mechanisms.

Our models could be enriched in the future. In particular, one of the key feature of the Cloud we do not support is elasticity. It would be interesting to study the integration of component-based approaches to tackle dynamic applications. For example, a broad class of applications spawns replicas of services depending on the load. Besides, a complete approach would be able to add or remove part of the application at runtime resulting in a completely different architecture.

Chapter 3

Formalization of Security Properties

In the previous chapter, we have presented a list of security properties a user can use to specify the security policy of a virtualized application. This policy defines what a secure system is *i.e.,* what this system is allowed or forbidden to do. But one question is left aside: *what is the meaning of a security property.* Previously, we have given general definitions in a natural language (*i.e.,* English) as formulated by standardization institutes (*e.g.,* Common Criteria, NIST). But as stated by Halpern *et al.* [57], security policies described in a natural language have quite ambiguous semantics. On the other hand, a formal language or logic provides clear syntax and semantics. By *formal*, we mean the specification is based on a strong mathematical foundation. The *syntax* is the grammar of the language, how a correct sentence is structured. The *semantics* is the meaning of the sentence. For example, "He was the first" and "He was the last" have the same structure (*i.e.,* subject, verb, noun) but not the same meaning.

Our goal is to use existing (maybe future) protection mechanisms to enforce information flow properties. It implies our formal language must be expressive enough to map a property to a configuration of a protection mechanism. However, the security properties modeled using Sam4C are global whereas existing security mechanisms are mostly local. For instance, it is easy to state that our house must be globally secured but this statement shall be implemented with local solutions such as putting locks on doors, installing an alarm, etc. Similarly, the most important application of our logic is to prove the equivalence between a global property and a set of locally enforceable properties. This transformation, occuring during the preprocessing step of our approach, is detailed later in Section 4.1.

In this chapter, we first lay the foundation for understanding advanced logic notions. Then we review related work on logics for security properties. After, we present our logic called IF-PLTL to specify information flow properties for concurrent and distributed systems. Finally, because IF-PLTL is strictly more expressive than any existing mechanisms, we give a complete algorithm to dynamically monitor any IF-PLTL formula. This monitor can be viewed as a potential "ideal" mechanism but due to its complexity (in space and time), an implementation of this monitor would incur an high overhead.

3.1 Logic 101

This section is intended to help readers with no logic background to grasp basic notions and go through the rest of this chapter more handily. For the ones that have such background, you can skip this section.

First, many definitions of the term logic have been given and it seems the debate is still going on. Instead, we prefer to present three principles of logic (or guidelines) as defined by Ferreiro in [47]:

1. Logic is concerned with an analysis of (valid and invalid) deduction.

2. Logic does not depend on how things are, it is independent of considerations of existence. Logic should enjoy universal applicability.

3. Logic only studies the form of arguments and deductions, never the matter. The basic idea was embodied in the use of variables A, B by Aristotle.

Formal logic encompasses a wide variety of logic systems from Aristotle's classical logic to modern systems emerging in the mid-19th century.

3.1.1 Syllogistic or Classical Logic

Syllogic or Classical Logic refers to Aristotle logical system based on syllogism. A syllogism is a two-premise deductive argument in which a conclusion is inferred from two premises. Let's take an example:

```
All men are mortal
Socrates is a man
Therefore, Socrates is mortal
```

The two first sentences are *premises* and the third the *conclusion*. This construction is called an *argument*. The premises and the conclusion are called *propositions*. What is important is to decorrelate the notion of *Truth* from the argument itself. First, we must consider the construction (the form) of the argument to determine if it is *valid*. An argument is *valid* if considering the premises are true then the conclusion must be true. But it does not tell us if the premises are true or false, just if you admit them, you must admit the conclusion. Classical logic is founded on three laws called *axioms*:

- The identity: If a proposition is true, then it is true.

- The excluded middle: A proposition is either true or false. There is no other possible outcome.

- The non-contradiction: A proposition cannot be both true and false.

3.1.2 Propositional Logic

Some logicians found Classical logic to be limited: indeed not every argument could fit in 3 lines. So they introduced new symbols, the most common being: negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), conditional ($\rightarrow$) and equivalence ($=$). These are connectives: they connect propositions to make more complex propositions. An *atomic* proposition is usually a single letter *e.g.,* p, q. Then, a complex proposition could be: $\neg(p \vee q)$.

In Classical logic, determining a valid or invalid argument is based on forms (*i.e.,* patterns). In propositional logic, there are no forms, instead we can rely on *Truth tables* (but not only) to analyze all permutations of true and false for every atomic proposition. For example, suppose these two statements $\neg(p \vee q)$ and $(\neg p \wedge \neg q)$ to be logically equivalent. By dressing the truth table, you will see that whenever the first is true, the second is also true and whenever the first is false the other is false as well. This is one of De Morgan's theorems and it is a *tautology*: the logical equivalence is true whatever the truth-values of p and q. In propositional logic, an argument is valid if and only if its conditional form (*i.e.,* using →) is a tautology.

3.1.3 Predicate or First-Order Logic

Propositional logic is not suited to formalize some valid arguments. The previous example `All men are mortal;Socrates is a man;Therefore, Socrates is mortal` in propositional logic yields $p \wedge q \rightarrow r$. Because, there are no relations between the 3 atomic propositions (p, q and r), this argument is not valid. For instance, we can assign truth-values to p, q and r to obtain $true \wedge true \rightarrow false$ which is intuitively wrong. In predicate logic, propositions are enriched with *predicates i.e.,* boolean functions. For example stating that Socrates is a man would be $Man(socrates)$ with Man a predicate and *socrates* an individual. In addition, logicians introduce the notion of *quantification* with "for all" ($\forall$) and "it exists"($\exists$).

Now, the three statements of our example can be expressed as follows: $\forall x Man(x) \rightarrow Mortal(x)$, $Man(socrates)$ and $Mortal(socrates)$.

Individuals are referred to using a *constant* (*e.g., socrates*) or a *variable* (*e.g.,* x). The difference between the two is that $Man(socrates)$ is either false or true but the truth-value of $Man(x)$ depends on the individual represented by x. The replacement of x by a constant is called an *assignment.* Hence, the truth-value of $Man(x)$ only exists after assignment. In addition to constants, variables and predicates, predicate logic may contain *functions.* For example, the statement $2 + 2 = 4$ may be formalized using a function *Plus* and a predicate *Equals*. It can be written as $Equals(Plus(2, 2), 4)$.

Predicate logic is a generic term encompassing first-order (and higher) logics, modal logic and others. The set of elements under discussion is called the *domain of discourse.* It can be the set of natural numbers, the set of species on Earth or the set of processes in a system. In first-order logic, variables (*i.e.,* x) can only range over these elements *e.g.,* "It exists an individual x who have donated more than 100$ to charity" is $(\exists x)(Donation(x) > 100)$. In second-order logic, quantifiers may range over sets of functions and predicates as well *e.g.,* "It exists an organization O such as all organization's members have donated more than 100$ to charity" is $(\exists O)(\forall x \in O)(Donation(x) > 100)$. Second-order logic is strictly more expressive than first-order logic and cannot be reduced to first-order logic. It means that a second-order formula cannot be transformed into a first-order formula (*e.g.,* by adding relations).

3.1.4 Modal Logic

In 1918, Clarence Irving Lewis introduced a formal modal logic system. Modal logic integrates the modalities of *necessity* and *possibility*. Modalities allows expressing propositions such as "It will rain today" (necessity) or "It might rain today" (possibility). A statement can be possible but not true. To clearly picture this notion, it can be viewed as *possible worlds* (or alternate universes). Then, the necessity modality is true if the related proposition is true in *every* possible world and the possibility modality is true if the proposition is true in *at least one* world.

Modal logic includes deontic and temporal logic.

Deontic Logic

In deontic logic, the modalities are related to the concept of *obligation*. "You must do this" and "You may do this" are close to necessity and possibility. One particularity of deontic logic is the concept of *necessitation*. This property says that whatever is necessitated by a moral requirement is itself a moral requirement. Suppose that you are morally required to break a door to save someone surrounded by fire. You are logically necessitated to break the door in order to save the person. The necessitation property tells you that you are obliged to save this person so you have an obligation to break the door. This is contestable because taken independently you have no obligation to break this door. This principle is the reason for many paradoxes in deontic logic [58].

Temporal Logic

In temporal logic, the modalities are related to the notion of *time*. A proposition is qualified by when it occurs, for example "It will eventually rain", "It will always rain" or "It has rain yesterday". In 1977, Pueli [98] proposed using temporal logic to formalize the behavior of concurrent programs of a system. He expressed properties like the mutual exclusion *i.e.,* two processes cannot enter the same critical section. To do so, he models the system as a state-transition system: an execution is a sequence of transitions (*e.g.,* actions, commands) where a transition changes the state of a system to another. He introduced two operators F and G to express respectively that something will eventually happen and something will always happen. In our previous example, "It will eventually rain" is captured by $F(rain)$ and "It will always rain" by $G(rain)$. Later, new operators have been introduced to discuss past events, namely P (once in the past) and H (always). For instance, "It has rained one day" is captured by $P(rain)$ and "It has always rained" by $H(rain)$.

3.1.5 Verification of Logics

One interest of logics is *verification*: showing that a system satisfies its specification. A system is described as a model and the specification is a formula of a formal language.

We consider two types of verifications: *model-checking* [33] and *runtime verification* [10].

In *model-checking*, all executions of a given system are examined to answer whether these satisfy a given property (*i.e.,* formula). The verification against all possible executions is typically been carried out by generating the whole state space of the underlying system, which often becomes unfeasible due to its huge size. Model-checking suffers from the state explosion problem.

In *runtime verification*, we examine a single execution, the one currently under scrutiny. The objective is to check if the current execution satisfies (or violates) a given property. Runtime verification does not suffer from state-explosion as there is no systematic exploration of multiple executions. This verification can be achieve offline by replaying recorded traces or online by making decisions as events occur. In the rest of this manuscript, we always refer to online runtime verification.

3.2 State of the Art

In this thesis, we propose a formalization of information flow security properties. We recall that Information Flow Control focuses on the propagation of information in a system. In this section, we review related work on formalization of security properties and present important information flow properties.

3.2.1 Information Flow Control properties

The main question is what kind of properties are needed to control information flows or rather what properties *can* we consider.

Trace properties

Security properties are properties of the system's behavior. In [75], Lamport has shown that a concurrent system's behavior can be modeled by all execution traces generated by this system where an execution trace is a sequence of state transitions.

It follows that a security property can be defined as a property on individual traces called **traces properties**. In [1], Alpern and Schneider have proved that a *trace property* is an intersection between a *safety* and a *liveness* property. The safety property ensures that nothing bad will happen and the liveness that something good will always happen. The *safety* is quite straightforward: a security policy forbid any "bad" behavior of the system. The *liveness* is more delicate to apprehend. It is somewhat close to the concept of availability. While being protected, it means the system keeps delivering its service as it should. As the availability, we believe it has more to do with reliability and quality of service than strict protection. Proposed by Pnueli in [98], Linear Temporal Logic (LTL) is a well-known formalism to express safety and liveness properties in concurrent systems.

Hyperproperties

It has been pointed out that some important security policies could not be captured by traces properties [35]. For example, stipulating a bound on mean response time over all executions is an availability policy that cannot be specified as a property of individual traces. Each execution response time is conditioned by the others. Another property is the *noninterference* introduced by Goguen and Meyers [51]. This confidential property stipulates that commands executed by a user with high-level clearance has no observable effects for users with low-level clearance. Noninterference has been revisited by Haigh and Young [56] then Rushby [105]. Noninterference would seem to be a fundamental notion in information security. It arises from the need to understand why a program would leak some bits of information. But noninterference seems to characterize the absence of information flows when in reality we want to control flows not forbid any of them. We discard noninterference as a system-level property based on the following argument stated in [106]:

> *Most "real" security policies are concerned with specifying who has access to what resources under what circumstances. Non-interference is never mentioned. Furthermore, non-interference is in practice impossible to realize in any real system: contention for resources, etc. render it unfeasible.*

In [35], Clarkson and Schneider have described these properties as *hyperproperties.* They observed that there is a vast range of security properties that cannot be stated on individual traces but rather on sets of traces. For example, noninterference typically involves 2 traces (*i.e.,* high and low) and the mean response time property applies to k traces. Informally, an *hyperproperty* is a property on *sets of traces* while a *trace property* is a property on *traces.* Hyperproperties are considered as really powerful. They are more general than security properties (in the sense of confidentiality and integrity); they allow making statements on quality of service for instance.

Now, the question is whether modelchecking and runtime verification of hyperproperties are *decidable.* A problem is said decidable if there exists an effective method to give an answer (*e.g.,* yes or no) in finite time to this problem. Therefore, providing an algorithm for modelchecking of hyperproperties implies that it is decidable. Clarkson and Schneider noted that the full power of second-order logic is necessary to express hyperproperties and thus it does not exist any general methods to verify an hyperproperty [35]. However, considering a fragment (*i.e.,* subset) may be decidable. Thus, in [34], Clarkson *et al.* have proposed temporal logics for some hyperproperties called HyperLTL, HyperCTL and HyperCTL* and detailed a modelchecking procedure to verify them.

We recall that modelchecking is a static method that necessitates a model to represent (or generate) all executions. For hyperproperties, it must represent all sets of all executions. This state explosion problem is an impediment to tackle existing production systems.

More importantly, in his thesis "Reasoning about hyperproperties" [87], Milushev discusses the possibility of dynamic enforcement of hyperproperties: even if some enforcement methods have been proposed for single specific hyperproperties, decidability of dynamic enforcement of broader classes is still an open question.

Nevertheless, we argue that HyperLTL (and their extensions) cannot tackle runtime verification for security properties. The argument is simple. Consider the following example of a trace property, *the reachability problem.* It is defined as: *Does it exist an execution (a trace) where a given state is reached i.e.,* it appears in the trace. It is clear that such trace property is satisfiable in the static case *i.e.,* with complete knowledge of states of the system but it is not the case for runtime verification. In the latter, the question must be answered at a time when a trace only represents what *has* occurred but never what *will* occur. This problem has been circumvented by Bauer in [10] when monitoring LTL formulas at runtime by introducing a three-state output: *true*, *false* or *inconclusive*. Applied to security, it would mean that if one requests access to a file, the answer is either "yes", "no" or "cannot decide yet" which is not acceptable.

With dynamic traces (*i.e.,* runtime verification), it is generally impossible to know in advance the next state of the system or make decisions based on the future.

In conclusion, we will focus on security properties verifiable at runtime *i.e.,* trace properties.

3.2.2 Logic-based Policies

Access Control Logics

Access control may be viewed as a simple trace property where the decision is generally independent from previous or future states of the system.

Many work use logic to model access control. Halpern *et al.* [57] apply First-Order (FO) logic to digital rights management. Datalog [63] is a logic programming language oftenly used to query databases. It is reducible to FO logic. Cassandra [11] is a role-based trust management system based on Datalog with constraints. It specifies AC policies for large-scale systems. Binder [42] also uses and extends Datalog to express distributed security statements. It can express operations like certification or delegation. Bruns *et al.* [26] propose a specification in Belnap logic for analyzable AC policy composition. Finally, the authors of [6] specify and implement *(Temporal) Role-Based Access Control* policies in *constraint logic programming.* Despite that all previously cited works exclusively tackle access control models (and not IFC), they are very interesting for their strong formalism allowing static analysis and modelchecking *e.g.,* for Operating System policies analysis [55]. Again, the issue with modelchecking approaches is the need to represent a complex system with an automaton-like model. Indeed, to be able to do so, the system must be a whitebox. Moreover, if the application/system changes, then the automaton changes as well. This leads to redo all analyses. Putting aside the question of the model size, this modeling constraint does not seem to suit current virtualized architectures where dynamicity and multitenancy are key-features.

Direct and Indirect Flows

Many attack patterns circumvent access controls by making *indirect accesses*.

As depicted in Figure 3.1, information may be transferred either *directly* or *indirectly* from an entity to another. Suppose the direct flow A to B is followed by the flow B to C then it creates an indirect flow from A to C. If the flows were to happen in reverse order (*i.e.,* B to C, then A to B), it would not have created an indirect flow as the information of A would have been carried only to B and not relayed thereafter. An indirect flow can involve a sequence of direct flows of arbitrary length.

Indirect flows are important as they allow controlling the path of propagation of information.

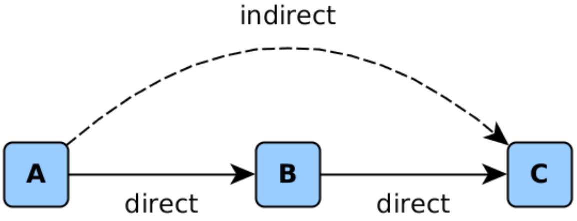


Figure 3.1: Direct and indirect flows

In this chapter, we are interested in expressing properties depending on previous actions. One of our contributions is to consider *all indirect paths* resulting from entities interactions instead of controlling indirect flows by construction.

To our knowledge, the closest work from ours has been proposed by Basin *et al.* in [9]. The authors propose a runtime monitoring with a Metric First-Order Temporal Logic (MFOTL) and provide algorithms to enforce formulas. Their approach is similar in the way that MFOTL includes the Linear Time Logic temporal modalities and the system behavior is modeled as (sets of) traces. Our formalization is noticeably inspired from this work. Nonetheless, the authors' proposal differs from ours in the language expressiveness. Indeed, there is a class of properties expressible with our approach and not with MFOTL.

In short, the indirect flow relation is the *transitive closure* of the *flow to* relation (*i.e.,* direct flow). Without the transitive closure, it is not possible to express an *unbounded* chain of flows with a *finite* formula. Suppose Alice wants to transfer information to Bob and to do so communicates through a chain of intermediates. Without the transitive closure relation, forbidding this indirect communication would necessitate specifying every possible chain of intermediates of any length. This is an infinite enumeration thus not expressible. Even with a bounded chain of flows, the formula's size is linear in the size of the chain. More generally, both MFOTL and our approach is reducible to First-Order (FO) formulas. Ronald Fagin [46] has shown that first-order logic extended with Transitive Closure (TC) is strictly more expressive than FO. It's called FO(TC) and yields NL *i.e.,* the problems solvable in nondeterministic logarithmic space. In consequence, we choose to trade more expressiveness for more complexity.

3.2.3 Discussion

Our goal is to formalize traditional security properties such as confidentiality or integrity as information flows. This formalization must encompass direct and indirect flows, and be strictly more expressive than most security mechanisms. The idea is to have a mapping between a security mechanism and a formula in our logic without targeting specific security mechanisms. Nonetheless, we limit our scope of mechanisms to runtime verification (*i.e.,* monitors) in opposition to modelchecking which is an offline analysis.

Multiple formal systems have been proposed to reason about security properties. First, there is not a single definition of a security property. We have presented two formulation mainly adopted: trace properties and hyperproperties. Hyperproperties are too expressive for our protection point of view *e.g.,* they allow for statements on quality of service. Moreover, there are serious doubts on the possibility of monitoring hyperproperties at runtime. On the other hand, trace properties can be monitored at runtime and existing security mechanisms enforce trace properties though they enforce simple ones.

As we focus on system-level information flow, it seems unrealistic to modify applications or assume a complete and perfect representation of them (like it would be needed for modelchecking). Instead, we consider applications to be blackboxes *i.e.,* we do not know their internal operating or logic. However, we suppose to have a partial knowledge *i.e.,* though applications are blackboxes, we can capture their system-level operations (*e.g.,* read, write) and we know how they ought to interact with other entities. Let's illustrate what we call partial knowledge with a simple example drawn in Figure 3.2. Two users, Alice and Bob, are using the same application hosted on a server. We want the following information flow property:

Alice and Bob are isolated from each other *i.e.,* information cannot flow from Alice to Bob (and vice versa).

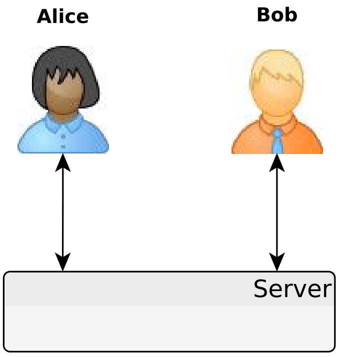


Figure 3.2: Alice and Bob must not exchange information with each other.

From a specification point of view, this system-wide isolation property is expressed on a blackbox *i.e.,* without any knowledge of the application. As a consequence, our approach is *specification-driven* in opposition to *configuration-driven* approaches where the property is realized by a thorough configuration of credentials, labels or tags. From an enforcement point of view, information flow events must be interposed to either allow or deny them with respect to their consequences. In short, any event leading Bob to have information from Alice must be denied.

Considering the gap in related work to express information flows with our requirements, in particular the transitive closure, we propose a new logic called Information Flow Past Linear Time Logic (IF-PLTL). Our logic is based on the past fragment of LTL (PLTL) which is known to be suited for describing the behavior of a concurrent system.

3.3 Overview

In the rest of this chapter, we detail our formalization of information flow properties namely IF-PLTL. Even though our goal is to encompass a vast range of protection mechanisms, we present our logic as an "ideal" protection mechanism by providing a monitoring algorithm to verify properties at runtime in a concurrent system (*e.g.,* an operating system) as shown in Figure 3.3.

As we have stressed in Section 3.2, it is mandatory to take into account *direct* and *indirect* information flows to control the propagation of information. But a system like an operating system do not have any abstraction to capture information flows directly. It is why we explain in the next section how to reconstruct information flow traces (including indirect information flows) from low-level *System Events.* In Figure 3.3, this process is represented as the *Trace Computation* box. It receives *System Events* from the *Events Interposition* box and produces *Information Flow Events.* How to implement an *Events Interposition* mechanism is quickly discussed in Section 3.4.1.

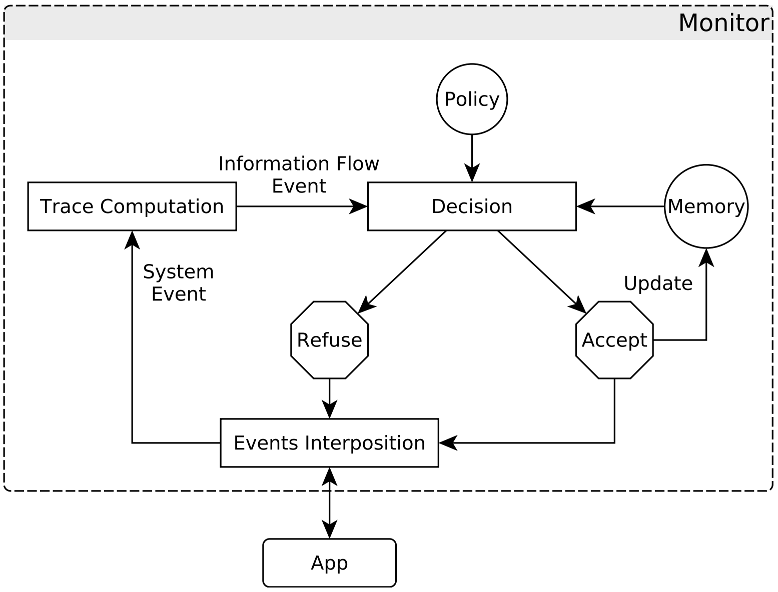


Figure 3.3: Monitor Architecture

The *Information Flow Event* is received by the *Decision* box. But it is not always possible to reason on a single *Information Flow Event.* Indeed, it would mean taking a decision without any knowledge of the past decisions (and the related information flows that have happened). Consequently, the *Decision* box takes a second input: the *Memory i.e.,* IF-Trace. Its formalism is described in the next section. On top of the *Information Flow Event*, this box also receives as input a *Policy.* In Section 3.5, we present our logic and its semantics allowing expressing such policy. Moreover, as we need to reason on past events, the logic is based on PLTL that brings such ability without the cost of too much complexity. Our logic allows expressing global security property with partial knowledge. It means that a user can express its security requirements without the need of a fine grained knowledge of the underlying systems and applications (as explained in Section 3.2.3).

The *Accept* and *Refuse* octagons in Figure 3.3 reflect the fact that the *System Event* (and the related *Information Flow Event*) satisfies (or not) the *Policy.* In the first case (*Accept*), the event is allowed to happen and the *Memory* is updated with the related information flow to keep track of it. If the event does not satisfy the policy, the event is not allowed and thus does not happen. Accordingly, there is no need to update the *Memory.* We describe how our logic can be used to implement such dynamic monitoring in Section 3.6. In Section 3.7, we conclude by presenting how our approach can be used to enforce a simple yet widely spread security requirement: the isolation between 2 users (or groups of users).

To summarize, our proposal can be classified as a System-based Information Flow Control mechanism. At the best of our knowledge, it is the only System-based Information Flow Control with a strong theoretical logic that considers all the direct and indirect information flows.

3.4 System Model: Traces Acquisition

To recapitulate, our goal is to decide whether the system's behavior modeled as traces *satisfies* a security policy using logic formulas as formal representation. In this section, *Trace Computation* part in Figure 3.3 details how to construct *information flow* traces from low-level system events.

As shown in Figure 3.4, a system (*App*) does not directly produce *information flows* but low-level *observable events* instead. These events are transformed into *functional events* and finally into *information flows*.

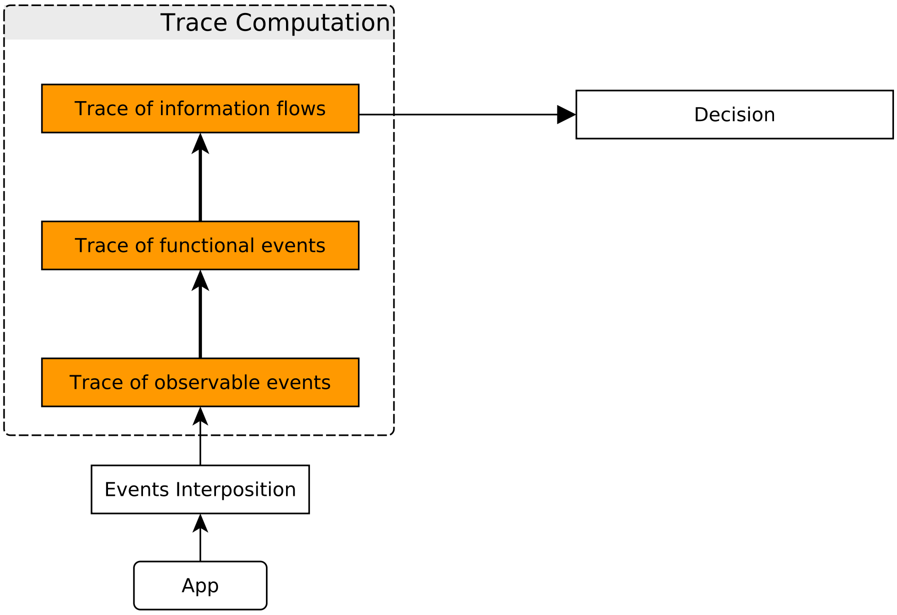

Trace Computation
Trace of information flows
Decision
Trace of functional events
Trace of observable events
Events Interposition
App

Figure 3.4: Overview of the computation of traces

Let's first propose the following definitions:

Definition 1 (Primitive System)**.** *A blackbox with an internal state τ and an interface accepting a set of actions.*

Definition 2 (System)**.** *A (recursive) composition of primitive systems.*

System behavior as traces

Every entity inside the Operating System (OS) environment can be viewed as systems *e.g.,* files, processes, databases. As previously said, a concurrent system behavior can be modeled by all execution traces generated by this system [75], where an execution trace is a sequence of state transitions. We give afterwards a general definition (Definition 3) of traces including the concept of *observer* (Definition 4). Let's suppose a system with only three entities (a,b,c). a has a partial trace including all events from or to itself but does not see events between b and c. Moreover, an event from a to b is both in a and b traces. As a result, the union of all partial traces forms a complete trace of the system.

Definition 3 (Trace). *A sequence of state transitions triggered by events, as viewed by an observer.*

Definition 4 (Observer). *A passive entity with a system view (potentially partial).*

Contexts

As entities are heterogeneous, they are identified using *contexts* (Definition 5). We suppose the existence of a method to map a context to every entity, where two entities with the same context have a set of common characteristics *e.g.,* behavior, security domain, type. These expressiveness of the contexts can be used to mimic the labels and tags proposed in other approaches. Nevertheless, the contexts are as expressive as the system allows and as it is required to enforce the security. Indeed, with finer-grained contexts, the security policy can specify finer-grained properties. For example, a context *file-editor* could be associated to the path `/usr/bin/vim` and/or `C:\Windows\bin\notepad.exe` In the following, we note $\mathcal{SC}$ the set of all contexts.

$\mathcal{SC}$: The set of all contexts.

Definition 5 (Context). *An abstract concept to refer to entities with a (set of) common characteristic(s).*

3.4.1 Traces with Observable Events

The execution platform has a complete view of all traces of the system. For the sake of clarity, let's take the example of processes running on top of the OS kernel. System calls (executed in kernel space) *e.g.,* `sys_read`, `sys_write`, `sys_fork`, are made from an entity to another one (potentially newly created in the fork case). In current Linux kernels, the Linux Security Module (LSM) [121] provides all needed kernel hooks to generate system traces from system calls. However, these hooks only occur before system calls and cannot see whenever a call ends. These pre-call hooks are not sufficient to represent the duration of an event and *a fortiori* the concurrency between two overlapping calls. In the following, we suppose to be able to capture *begin* and *end* events of any call. One should note it is possible to implement such module in any Linux kernel [19]. The same could be done for network flows by creating a netfilter plugin[1].

An *observable event* (Definition 6) is an atomic event viewed by an observer *e.g.,* the kernel. It is a triple (a, eop, b) where a, b are contexts and eop is an elementary operation, for instance $(alice, begin\_read, logs)$.

$\mathcal{EOP}$: set of elementary operations (begin_read, ...)
$\mathcal{OE}$: set of observable events $(\mathcal{SC} \times \mathcal{EOP} \times \mathcal{SC})$

Definition 6 (Observable Event)**.**

$$oe \in \mathcal{OE} \equiv_{def} (a, eop, b) \textbf{ where } \begin{cases} a, b \in \mathcal{SC} \\ eop \in \mathcal{EOP} \end{cases}$$

The low-level trace (Definition 7) produced by the system is a set of observable events. Figure 3.5 shows an example of trace for a system composed of three entities (a, b, c), a read operation has finished and a write operation is still occurring (no ending event).

Definition 7 (Trace of observable events)**.**

$T \ \equiv_{def} \ \{oe_1, oe_2, \ldots, oe_n\}$ **where** $oe_i \in \mathcal{OE}$

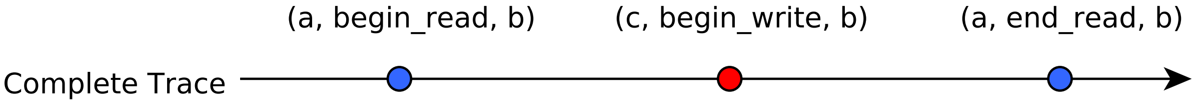


Figure 3.5: Trace of observable events

3.4.2 Traces with Functional Events

Observable events are *discrete* events: they occur at a precise point in time. Nevertheless, read and write operations programmers are familiar with, last for a certain period of time: they are *continuous* events we call *functional events.*

With *begin* and *end* events, we can build the corresponding *functional event* (Definition 8) including the *functional operation e.g.,* read, write. An entity may perform multiple functional operations in parallel. Let's suppose a context a is reading twice a context b *e.g.,* two entities with the context a read a file with the context b. To correctly associate the *end* event and the *begin* event, an observer must discriminate the two parallel read operations even if the sources and the destinations are identical. Therefore, we suppose two functions is_begin_event and is_end_event, determining respectively if an observable event oe_i is the beginning of a functional operation op and if an observable event oe_j is the ending of a functional operation op starting at oe_i. For example, suppose $eo_1 = (alice, begin\_read, logs)$. It is the begin event of a *read* but $eo_2 = (alice, end\_read, logs)$ may be the end event of eo_1 *read* operation.

All events in the functional trace (Definition 9) are in the set $\mathcal{FE}$ and all operations in the set $\mathcal{FO}$.

[1] `http://www.netfilter.org/`

$\mathcal{FO}$: set of functional operations (read, write, ...)
$\mathcal{FE}$: set of functional events ($\mathcal{SC} \times \mathcal{FO} \times \mathcal{SC}$)

The two functions *is_begin_event* and *is_end_event* are defined as follows:

is_begin_event: $\mathcal{OE} \times \mathcal{FO} \rightarrow \{true, false\}$
is_end_event: $\mathcal{OE} \times \mathcal{OE} \rightarrow \{true, false\}$

A functional event is a continuous event and thus considering a timeframe $[i, j]$, it occurs at every instant k if the begin event appears before or exactly at time i and the end event after or exactly at time j.

Definition 8 (Functional Event)**.**

$$\forall k \in [i,j], (a, op, b)_k \in \mathcal{FE} \equiv_{def} \begin{cases} a, b \in SC, op \in \mathcal{FO} \\ \exists i' \leqslant i, oe_{i'} = (a, eop_{begin}, b)_{i'} \wedge is\_begin\_event(oe_{i'}, op) \\ \exists j' \geqslant j, oe_{j'} = (a, eop_{end}, b)_{j'} \wedge is\_end\_event(oe_{j'}, oe_{i'}) \end{cases}$$

Definition 9 (Trace of functional events)**.**
$T \equiv_{def} \{fe_1, fe_2, \ldots, fe_n\}$ **where** $fe_i \in \mathcal{FE}$

Figure 3.6 is the result of projecting observable events of Figure 3.5 into functional events. In this trace, the first observable event $(a, begin\_read, b)$ and the third observable event (a, end_read, b) are transformed into a functional event $(a, read, b)$ occurring at every moment between the beginning and the end included.

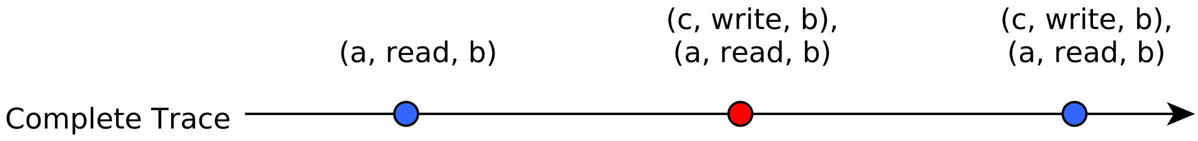


Figure 3.6: Trace of functional events

3.4.3 Traces with Information Flows

Information flow models differ by their expressiveness and their relations. Nonetheless, we can outline two common relations/operators:

- The flow-to relation *e.g.,* an information flows from a to b.
- The relabeling operator *e.g.,* an entity labeled with a is relabeled with b.

Relabeling is a classic operator allowing a user to change his identity (*e.g.,* su command in Linux). In terms of information flow, when transiting from label a to label b, the entity brings to b all the information he has as a. Therefore, there is a flow from a to b; the relabeling operation is classified as a write operation.

As a result, we consider only one relation in our information flow traces:

A flow from a to b is defined as $(a > b)$

To transform functional events into information flows, we introduce two functions determining if an arbitrary functional operation $(op \in \mathcal{FO})$ is equivalent to a *read* or a *write* operation:

is_read_like: $\mathcal{FO} \rightarrow \{true, false\}$
is_write_like: $\mathcal{FO} \rightarrow \{true, false\}$

The informal semantics of the previous operations are:

- a reads from b: information flows from b to a.
- a writes to b: information flows from a to b.

Definition 10 formally defines the relation $(>)$ using the functions *is_read_like* and *is_write_like*.

Definition 10 (Flow-to Relation)**.**

$$(a > b) \in \mathcal{IF} \equiv_{def} \exists op \in \mathcal{FO} \begin{cases} ((b, op, a) \wedge is\_read\_like(op)) \\ \vee \\ ((a, op, b) \wedge is\_write\_like(op)) \end{cases}$$

Figure 3.7 is the result of projecting functional events in Figure 3.6 into information flows. $(a, read, b)$ and $(c, write, b)$ are substituted by $(a < b)$ and $(c > b)$ respectively.

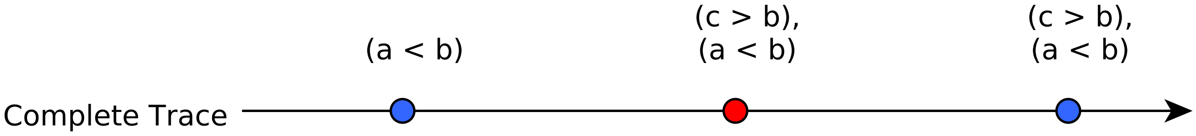


Figure 3.7: Trace of information flows

3.4.4 Summary

Instead of directly dealing with IF-traces, we have shown how to obtain them over more concrete traces. The two functions *is_begin_event* and *is_end_event* are sufficient to model continuous operations (functional events) over atomic ones (observable events). Then, with the two functions *is_read_like* and *is_write_like*, we have described how to finally obtain IF-traces.

The reader should note that despite we used the OS/kernel trace, the approach is not limited to this type of systems and has been historically applied to mobile systems (Android), JVM clusters and hypervisors [19] and could be applied to network flows, application services [48] or even to cryptographic primitives.

3.5 Security Properties: Information Flow Past Linear Time Logic

After presenting how to transform "real" low-level events into information flows, we can now reason solely on these flows.

In runtime verification, we consider a unique finite trace representing the history of the current execution. Our goal is to verify a vast range of *trace properties*. We adopt the same definition of trace properties (Definitions 11) as defined by Clarkson *et al.* in [35]. We recall that a trace property is expressed on traces that come from a single observer *e.g.,* the kernel.

Definition 11 (Trace Property)**.** *A set of infinite traces.*

3.5.1 Temporal Many-Sorted Logic with Information Flow

In order to model trace properties, we need a **many-sorted first-order temporal logic** on information flows.

Temporal A *temporal* logic implicitly defines the flow of time over which formulas are evaluated. It allows making statements on past events such as "Alice cannot get information from Charlie if Bob has previously sent information to him".

First-Order A *first-order* logic allows using quantifiers to express properties such as: *There is a context a from which flows are initiated.*

Many-sorted Finally, a *many-sorted* logic, as opposed to a *single-sorted logic*, allows defining several *sorts* of domains instead of an homogeneous domain of discourse over which a quantifier iterates. With single-sorted logic, in the formula $\forall x P(x)$, variable x can range over any value under a single domain of discourse. To make sense, this domain should contain entities of same "kind" *e.g.,* contexts. But, in practice, we need to represent logical groups of contexts called *domains e.g.,* assign Alice and Bob to groups like Users or Admins. One possibility would be to represent groups as predicates *e.g.,* $Admins(alice)$ would be true if Alice is an administrator. But then, iterating over domains and expressing properties such as "Any contexts of any domains other than Admins may not send information to Alice" would require the power of second-order logic which poses several decidability and complexity issues we would not detail here.

Instead, using multiple sorts easily allows us distinguishing *contexts* and *domains* without requiring to use a second-order logic. Quantifiers are *sorted* and iterate over a single sort *e.g.,* contexts with $\forall_{ctx} x$ or domains with $\forall_{dom} d$.

In the following, we first give general definitions of a many sorted *signature i.e.,* the non-logical symbols of a many-sorted logic, to further detail the concrete signature of IF-PLTL.

Let's describe a many-sorted signature (Definition 12).

Definition 12 (Many-Sorted Signature)**.** *A many-sorted signature is a tuple* $\Sigma = (S, C, F, P)$ *where:*

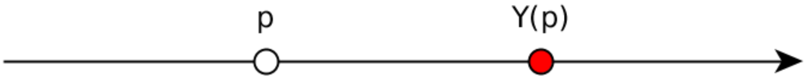
- $S = \{\sigma_1, \ldots, \sigma_n\}$ *where* $n > 0$ *and* σ_i *is a sort.*
- $C = \{c_1, \ldots, c_n\}$ *where* $n \geqslant 0$ *and* c_i *is a constant symbol of sort* $\sigma \in S$.
- $F = \{f_1, \ldots, f_n\}$ *where* $n \geqslant 0$ *and* f_i *is a function symbol of arity*[2] $m \geqslant 0$ *with sorts* $(\sigma_1 \times \ldots \times \sigma_m) \to \sigma$ *where* $\sigma_i \in S$ *and* $\sigma \in S$.
- $P = \{p_1, \ldots, p_n\}$ *where* $n \geqslant 0$ *and* p_i *is a predicate symbol of arity* $m \geqslant 0$ *with sorts* $(\sigma_1 \times \ldots \times \sigma_m)$ *where* $\sigma_i \in S$.

Past-LTL

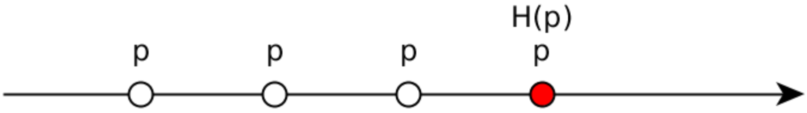
We can make implicit temporal information by using PLTL (Past-LTL) modalities [22, 98]. PLTL is a well-known formalism to express safety and liveness properties in concurrent systems. As a many-sorted logic is well-suited to specify IF properties, PLTL is well-suited to express *temporal* IF properties.

We would like to highlight that the goal is to decide if an event is allowed/forbidden according to past (and current) flows. Moreover, as explained before, we construct the trace as the system progresses *i.e.,* the *future* has yet to happen; any property on the future is unsatisfiable. Therefore, only pure-past PLTL modalities are satisfiable on dynamic traces. As a result, we consider the following PLTL modalities:

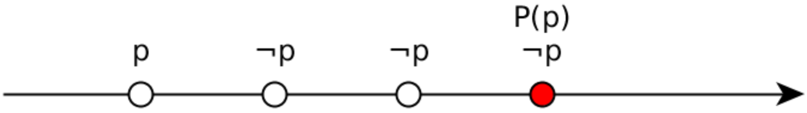
1. $Y(\varphi)$ **(Previous)**: φ had to hold at the previous state.



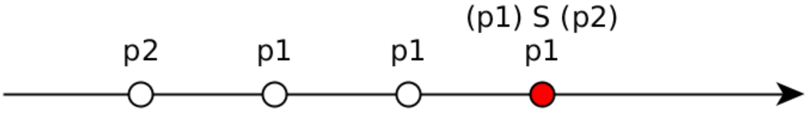
2. $H(\varphi)$ **(Globally in the past)**: φ had to hold on the entire subsequent path.



3. $P(\varphi)$ **(Eventually in the past)**: φ eventually had to hold (somewhere on the subsequent path).



4. $(\varphi_1)S(\varphi_2)$ **(Since)**: φ_1 had to hold since φ_2 held.



[2] The arity of a symbol is the number of "parameters" it takes

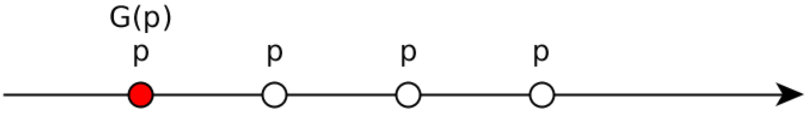
The modalities P and H are derived modalities and can be defined from the S modality as follows:

$$P\varphi \equiv \top S\varphi$$
$$H\varphi \equiv \neg P \neg \varphi$$

In the rest of this chapter, we will only use the past modalities S and Y as others can be rewritten into S-only formulas.

Although only pure-past modalities can be used in formulas, a security policy must be true at any point in time from the initialization of the system. Accordingly, we add LTL modality G:

- $G(\varphi)$ **(Globally in the future)**: φ has to hold on the entire subsequent path.



As a result, a policy Ψ is defined as $G(\varphi)$ where φ is a closed pure-past IF-PLTL formula. A *closed formula* (also ground formula) is a formula without any free variables. For instance, $P(x)$ is *not* closed as x is a free variable *i.e.,* there is no quantifier preceding x whereas $\forall x P(x)$ is a closed formula. A free variable is not quantified and can take any values.

3.5.2 IF-PLTL Syntax

The syntax describes how to construct *well-formed* formulas of our logic. A formula is *well-formed* when it is part of our formal language. In other words, a non-*well-formed* formula cannot be interpreted by our system: the formula has no meaning.

We first give the signature for temporal information flows and then describe the formation rules. To define the set C of constants, we use $\mathcal{SC}$ the set of contexts and introduce $\mathcal{SD}$ the set of domains.

Signature

$\Sigma_{\mathcal{IF}-PLTL} = (S, C, F, P)$

$S = \{ctx, dom\}$

$C = \mathcal{SC} \cup \mathcal{SD}$

$F = \varnothing$

$$P = \begin{array}{ll} \{>, \gg & : ctx \times ctx \\ \in & : ctx \times domain \\ \in & : domain \times domain \\ \approx & : ctx \times ctx \\ \approx & : domain \times domain\} \end{array}$$

We consider two sorts *ctx* and *dom* to designate *contexts* and *domains i.e.,* sets of contexts. Similarly to IF-traces, the flow-to relation ($>$) is naturally defined between contexts. We also introduce the *indirect flow* relation ($\gg$), the transitive closure of the direct flow-to relation ($>$); it is inferred from a sequence of direct flows. For example, the IF-trace in Figure 3.7 (Page 57) satisfies the property $c \gg a$ at the second and third moments because of the indirection $c > b$ and $b > a$.

The set membership relation ($\in$) is firstly defined between a context and a domain but also between domains to allow the specification of hierarchical sets. For example, suppose a context a, a domain Set and a domain $SuperSet$, we can define hierarchical relations where $a \in Set$ is true, $Set \in SuperSet$ is true and $a \in SuperSet$ is false.

Finally, we consider the equality relation ($\approx$) defined between either *contexts* or *domains*.

As we do not need any function, the set of functions F is empty.

Formation rules

Σ-TERM $t ::= x_\sigma \mid c_\sigma \mid f(t_1, \ldots, t_n)$ where $\begin{cases} x_\sigma \text{ is a variable of sort } \sigma. \\ c_\sigma \text{ is a constant symbol of sort } \sigma. \\ f : \sigma_1 \times \ldots \times \sigma_n \to \sigma \text{ is a function symbol} \\ \text{ with Σ-TERM } t_i \text{ of sort } \sigma_i. \end{cases}$

Σ-ATOM $a ::= p(t_1, \ldots, t_n)$ where $p : \sigma_1 \times \ldots \times \sigma_n$ is a predicate symbol with t_i Σ-TERM of sort σ_i.

Σ-FORMULA $\varphi ::= a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \exists_\sigma x \varphi(x) \mid Y(\varphi) \mid \varphi_1 S \varphi_2$ where a is a Σ-ATOM and x a variable of sort σ

Atoms and formulas can be enriched with the classical syntactic sugar, namely:

$$\begin{array}{l} \varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\ \forall_\sigma x \varphi(x) \equiv \neg(\exists_\sigma x \neg\varphi(x)) \\ (a \notin s) \equiv \neg(a \in s) \end{array}$$

We give some abbreviations for simplicity in formulas expression:

$$\begin{array}{l} (\forall x \in y)\varphi(x) \equiv (\forall x)\big((x \in y) \to \varphi(x)\big) \\ (\exists x \in y)\varphi(x) \equiv (\exists x)\big((x \in y) \wedge \varphi(x)\big) \end{array}$$

Well-formed formula

Let's give an example of a well-formed formula:

$$G((\forall_{ctx} a, b)(\exists_{dom} s)(a > b) \wedge (a \in s) \to (b \in s))$$

This formula states that at every moment of the execution, for any pair a, b of contexts, there is a domain s such as if information flows from a to b and a is in domain s, then b is also in domain s.

3.5.3 IF-PLTL Semantics

The semantic describes how to evaluate any well-formed formula of IF-PLTL. First, we define a FO many-sorted structure (Definition 13) obtained at every moment of the execution; there is no temporal notions in such structure. Next, we define a FO temporal structure (Definition 14) interpreting the flow of time. Then, we give the definition of the satisfaction relation ($\models$) between an IF-PLTL structure and an IF-PLTL formula.

A FO many-sorted structure $\mathfrak{M}'$ is composed of a domain $\mathcal{D}$ and an interpretation function $\mathcal{I}$. The domain is a set of all objects of sorts *ctx*, *dom*. The interpretation function defines the meaning of all symbols appearing in a formula (without temporal modalities).

Definition 13 (A First-Order Many-Sorted Structure).
A FO-many-sorted $\Sigma_{\mathcal{IF}-PLTL}$-structure (or model) is a tuple $\mathfrak{M}' = (\mathcal{D}, \mathcal{I})$ with:

1. $\mathcal{D} = \bigcup_{\sigma \in S} \mathcal{D}_\sigma$ *a many-sorted domain with $\mathcal{D}_\sigma$ a non empty domain.* $\forall \sigma_1, \sigma_2 \in S, \mathcal{D}_{\sigma_1} \cap \mathcal{D}_{\sigma_2} = \varnothing$.

2. $\mathcal{I}$ *an interpretation function over $\mathcal{D}$ satisfying the following properties:*

   (a) *Each sort $\sigma \in S$ is mapped to a non empty domain $\mathcal{D}_\sigma$.*

   (b) *Each constant symbol $c \in C$ of sort σ is mapped to an element $c^{\mathcal{I}} \in \mathcal{D}_\sigma$.*

   (c) *Each function symbol $f \in F$ of sorted arity $\sigma_1 \times \ldots \times \sigma_n \to \sigma$ is mapped to a function $f^{\mathcal{I}} : \mathcal{D}_{\sigma_1} \times \ldots \times \mathcal{D}_{\sigma_n} \to \mathcal{D}_\sigma$.*

   (d) *Each predicate symbol $p \in P$ of sorted arity $\sigma_1 \times \ldots \times \sigma_n$ is mapped to a subset $p^{\mathcal{I}} \subseteq \mathcal{D}_{\sigma_1} \times \ldots \times \mathcal{D}_{\sigma_n}$.*

A FO temporal structure $\mathfrak{M}$ is composed of a flow of time $\mathcal{F}$, the same domain $\mathcal{D}$ and a function $\mathcal{A}$ associating every moment of the execution to a FO many-sorted structure previously introduced.

Definition 14 (A First-Order Temporal Structure).
A FO-temporal $\Sigma_{\mathcal{IF}-PLTL}$-structure (or model) is defined by $\mathfrak{M} = \langle \mathcal{F}, \mathcal{D}, \mathcal{A} \rangle$ with:

1. $\mathcal{F} = \langle T, < \rangle$ *a strict linear order representing intended flow of time with $T \subseteq \mathbb{N}$.*

2. $\mathcal{D} = \bigcup_{\sigma \in S} \mathcal{D}_\sigma$ *a many-sorted domain with $\mathcal{D}_\sigma$ a non empty domain.* $\forall \sigma_1, \sigma_2 \in S, \mathcal{D}_{\sigma_1} \cap \mathcal{D}_{\sigma_2} = \varnothing$.

3. $\mathcal{A}$ *a function associating with every moment $k \in T$ a first-order many-sorted structure $\mathcal{A}(k) = \langle \mathcal{D}, \mathcal{I}^k \rangle$ with $\mathcal{I}^k$ the interpretation at moment k.*

We use the notation $(\mathfrak{M}, k)$ for the FO many-sorted structure at moment k.

The satisfaction relation (or truth-relation) $(\mathfrak{M}, k) \models \varphi$ between a model (structure) $\mathfrak{M}$ and a formula φ at the moment k is defined as follows:

$$\begin{array}{ll}
(\mathfrak{M},k) \models (a \approx b) & \text{iff } (a \approx b) \in \mathcal{I}^k(\approx) \\
(\mathfrak{M},k) \models (a > b) & \text{iff } (a > b) \in \mathcal{I}^k(>) \\
(\mathfrak{M},k) \models (a \in s) & \text{iff } (a \in s) \in \mathcal{I}^k(\in) \\
(\mathfrak{M},k) \models (a \gg b) & \text{iff } (\mathfrak{M},k) \models (a > b) \text{ or} \\
 & \qquad \exists i, (0 \leqslant i \leqslant k), \exists_{ctx} c, \begin{cases} (\mathfrak{M},k) \models (c > b) \\ \text{and} \\ (\mathfrak{M},i) \models (a \gg c) \end{cases} \\
(\mathfrak{M},k) \models \neg\varphi & \text{iff } (\mathfrak{M},k) \not\models \varphi \\
(\mathfrak{M},k) \models \varphi_1 \wedge \varphi_2 & \text{iff } (\mathfrak{M},k) \models \varphi_1 \text{ and } (\mathfrak{M},k) \models \varphi_2 \\
(\mathfrak{M},k) \models \exists_\sigma x \varphi(x) & \text{iff } (\mathfrak{M},k) \models \varphi[x^{\mathcal{I}}/x] \text{ for some } x^{\mathcal{I}} \in \mathcal{D}_\sigma \\
(\mathfrak{M},k) \models Y\varphi & \text{iff } 0 < k \text{ and } (\mathfrak{M},k-1) \models \varphi \\
(\mathfrak{M},k) \models \varphi_1 S \varphi_2 & \text{iff } \exists i, (0 \leqslant i \leqslant k), \begin{cases} (\mathfrak{M},i) \models \varphi_2 \\ \text{and} \\ \forall j, (i < j \leqslant k), (\mathfrak{M},j) \models \varphi_1 \end{cases}
\end{array}$$

Satisfiability

As previously explained, the existence of a future state is unsatisfiable for a dynamic trace (constructed as events occur). Here, we make two general observations:

Observation 1. *In the general case, a model $\mathfrak{M}$ at moment k can satisfy a theory T (set of closed formulas), but not at moment $k+1$.*
$(\mathfrak{M},k) \models T \;\not\Longrightarrow\; (\mathfrak{M},k+1) \models T$

Observation 2. *In the general case, a model $\mathfrak{M}$ would not satisfy a theory T at moment k, but could at moment $k+1$.*
$(\mathfrak{M},k) \not\models T \;\not\Longrightarrow\; (\mathfrak{M},k+1) \not\models T$

Let's clarify observations 1 and 2. Suppose a theory forbidding information to directly flow from an entity a to another entity b *i.e.,* $(a \not> b)$. If at moment k, a has never sent any information to b, then the theory is satisfied. Now, if a sends information to b at moment $k+1$, then the property is not satisfied, which concludes Observation 1. In the opposite, suppose a theory compelling information to directly flow from an entity a to another entity b that is $(a > b)$. If at moment k, a has never sent any information to b, then the theory is not satisfied. Now, if a sends information to b at moment $k+1$, then the property is satisfied, which concludes Observation 2.

Isolation

An important problem in distributed systems with a large number of users such as Clouds is the isolation between users. We propose the following definition of isolation:

Definition 15 (Isolation)**.**
Two entities A and B are isolated from each other in a system if there is no direct or indirect flow between A and B inside this system.

Applied to groups, this property can be translated in IF-PLTL as:

$$(\forall u_1 \in D_1)(\forall u_2 \in D_2)\neg(u_1 \gg u_2) \wedge \neg(u_2 \gg u_1)$$

Accordingly, to enforce the isolation between the two users of our example shown in Figure 3.2, we just need to specify 2 groups: D_{Alice} (that contains the Alice user) and D_{Bob} (that contains the Bob user).

3.6 Dynamic Monitoring

Even if our primary goal is to prove the equivalence between a global property and a set of local properties (see Section 4.1), a dynamic monitor can be designed to encompass the full expressivity of IF-PLTL with higher complexity in space and time than traditional mechanisms.

This section describes the *Decision* and *Memory* boxes of the IF-PLTL monitoring architecture depicted in Figure 3.3. To that end, a valuation operator $[\![.]\!]^k$ is introduced to better understand what data needs to be kept from time step to time step and then a full decision algorithm is given and its complexity discussed.

3.6.1 Memory

In order to evaluate if a system's trace satisfies a given IF-PLTL formula at a given moment k, some information from previous moments (before k) is required. Storing the complete trace up to moment k would be correct but would require a linearly increasing amount of memory as time passes. The present section introduces a more efficient way to store information.

A first thing to notice is that, according to the satisfaction relation (defined in Section 3.5.3), only three relations require information from the past: S, Y and $\gg$.

For example, suppose

$$\psi = (\forall_{ctx} x)Y(x > Alice) \wedge (Bob > x)$$

The satisfaction of ψ at moment k depends on moment $k-1$ due to the Y operator. More specifically, the satisfaction of ψ depends on occurrences of events of the form $x > Alice$ at moment $k-1$. Since there is no other occurrence of S, Y or $\gg$ in ψ, only these events are relevant to compute the satisfaction. All other events can safely be forgotten.

To generalize this observation, we introduce a valuation operator $[\![\varphi]\!]^{(\mathfrak{M},k)}$ which associates to a formula φ with free variables all the assignments for these variables which satisfy φ at moment k according to model $\mathfrak{M}$. We simplify this notation by writing $[\![\varphi]\!]^k$, the model $\mathfrak{M}$ being implicitly defined.

The valuation $[\![\varphi]\!]^k$ can take up to three forms:

- $\{[]\}$ the tautology;

- $\{[x_1 : \delta_1 \dots x_n : \delta_n]^*\}$ the list of valid assignment vectors (substitutions such as φ is true);

- $\varnothing$ the falsity.

The operators $\cup$, $\cap$ and $\overline{\{.\}}$ are defined as the usual sets operators, namely the set union, the set intersection and the set complement over domain of discourse $\mathcal{D}$. However, we introduce a small subtlety: the compact vector.

Let's suppose two sets of compact vectors (S_1, S_2) with $x_0, \dots, x_i, x_{i+1}$ free variables:

$$S_1 = \{\begin{bmatrix} x_0 : v_0 \\ \vdots \\ x_i : v_i \end{bmatrix}\} \text{ and } S_2 = \{\begin{bmatrix} x_{i+1} : v_{i+1} \end{bmatrix}\}$$

The free variable x_{i+1} does not appear in S_1 meaning that it is not a free variable in the formula whose valuations are S_1. Thus, any value of x_{i+1} in S_1 is also a solution. As a result, S_1 can be unfolded with all values $v^j_{i+1} \in |\mathcal{D}|$. The unfolded result S_1^{unfold} is given herebelow:

$$S_1^{unfold} = \{\begin{bmatrix} x_0 : v_0 \\ \vdots \\ x_i : v_i \\ x_{i+1} : v^0_{i+1} \end{bmatrix}, \begin{bmatrix} x_0 : v_0 \\ \vdots \\ x_i : v_i \\ x_{i+1} : v^1_{i+1} \end{bmatrix}, \dots, \begin{bmatrix} x_0 : v_0 \\ \vdots \\ x_i : v_i \\ x_{i+1} : v^{|\mathcal{D}|}_{i+1} \end{bmatrix}\}$$

The same *unfolding* operation can be applied on S_2 and thus the set operators apply without changing their semantics. For example, the *compact* results of $\cap$ and $\cup$ are:

$$S_1 \cap S_2 = \{\begin{bmatrix} x_0 : v_0 \\ \vdots \\ x_i : v_i \\ x_{i+1} : v_{i+1} \end{bmatrix}\} \text{ and } S_1 \cup S_2 = \{\begin{bmatrix} x_0 : v_0 \\ \vdots \\ x_i : v_i \end{bmatrix}, \begin{bmatrix} x_{i+1} : v_{i+1} \end{bmatrix}\}$$

This *compact* form is compliant with our tautology and falsity definitions.

And finally, *remove*(L, x) is the vector operator removing key x from vector L.

The rules to compute the valuation can be directly deduced from the IF-PLTL satisfaction relation:

$$[\![a \approx b]\!]^k = \{[a : x^{\mathcal{I}}, b : x^{\mathcal{I}}] \mid \forall x^{\mathcal{I}} \in \mathcal{D}_{ctx}\}$$

$$[\![a > b]\!]^k = \{[a : x^{\mathcal{I}}, b : y^{\mathcal{I}}] \mid \forall x^{\mathcal{I}}, y^{\mathcal{I}} \in \mathcal{D}_{ctx}, (x^{\mathcal{I}} > y^{\mathcal{I}}) \in \mathcal{I}^k\}$$

$$[\![a \in s]\!]^k = \{[a : x^{\mathcal{I}}, s : y^{\mathcal{I}}] \mid \forall x^{\mathcal{I}} \in \mathcal{D}_{ctx}, \forall y^{\mathcal{I}} \in \mathcal{D}_{dom}, x^{\mathcal{I}} \in y^{\mathcal{I}}\}$$

$$[\![\neg\varphi]\!]^k = \overline{[\![\varphi]\!]^k}$$

$$[\![\varphi_1 \wedge \varphi_2]\!]^k = [\![\varphi_1]\!]^k \cap [\![\varphi_2]\!]^k$$

$$[\![\exists_\sigma x \varphi(x)]\!]^k = \{remove(L, x) \mid L \in [\![\varphi(x)]\!]^k\}$$

$$[\![Y(\varphi)]\!]^k = [\![\varphi]\!]^{k-1}$$

$$[\![(\varphi_1)S(\varphi_2)]\!]^k = [\![\varphi_2]\!]^k \cup ([\![\varphi_1]\!]^k \cap [\![(\varphi_1)S(\varphi_2)]\!]^{k-1})$$

So far, all valuations at moment k depend only on moment k and valuations at moment $k-1$. This is a desirable property since we want to limit memory usage. The case $x \gg y$ is more difficult and building a rule for it straight from the satisfaction relation would require storing events further in the past. To avoid this, we propose to compute $[\![x \gg y]\!]^k$ from a reachability list. If $(a > b) \in \mathcal{I}^k(>)$ *i.e.,* event $(a > b)$ occurs at k then:

$$\forall x,\ \textbf{if } a \in \mathit{reach}(x) \textbf{ then } \mathit{reach}(x) \leftarrow \mathit{reach}(x) \cup \{b\}$$

As a result, $(x \gg y)$ valuations are defined as:

$$[\![x \gg y]\!]^k = \{(x,y) \mid y \in \mathit{reach}(x) \wedge \exists z, (z > y) \in \mathcal{I}^k(>)\}$$

Overall, it is possible to compute satisfaction of a formula from an up-to-date reachability list and from up-to-date valuations of every subformula ψ such that $Y(\psi)$ or $S(\psi)$ appears in Ψ. We denote $\mathit{subf}_m(\Psi)$ this set of subformulas which is the set of all formulas to be monitored.

Memory-wise, this is a more efficient approach than storing all events. Its complexity is discussed further in Section 3.6.3.

3.6.2 Monitoring Algorithm

This section presents the decision algorithm itself and how memory is updated. Let's recall that decision occurs within the architecture presented in Figure 3.3. More precisely, the *Decision* box receives information flow events one-by-one and is responsible for accepting or rejecting them and for updating the memory.

Let's assume that the policy to be enforced is $G(\Psi)$ where Ψ is a closed formula.

Initialization

At startup, the list of subformulas whose valuations are to be computed is established. As we have seen in the previous section, this list comprises the subformulas of $\mathit{subf}_m(\Psi)$. This list is initialized at startup as a list of vectors of valuations and is denoted val_{pre}. The initial (empty) reachability list is also initialized at startup and is denoted reach_{pre}.

During Execution

Upon reception of an information flow event, the *Decision* box of our monitoring architecture executes the following algorithm:

```
Variables: $reach_{pre}, val_{pre}, \Psi$
  upon event $(x > y)$ do
    $reach_{new} \leftarrow$ ADDREACH($reach_{pre}$, $x > y$)
    INIT($val_{new}$)
    for all $\varphi \in$ subf($\Psi$) do
      {Monitored subformulas ordered by depth-inverse}
      $val_{new}[\varphi] \leftarrow$ COMPUTEVAL($\varphi$, $val_{pre}$, $val_{new}$)
    if SAT($\Psi$, $val_{new}$) then
      UPDATE($reach_{new}$, $val_{new}$)
```

```
        ACCEPT($x > y$)
    else
        REFUSE($x > y$)
```

Upon reception of an event, it is not yet known if this event is going to be accepted or not. Thus, the algorithm cannot update val_{pre} and $reach_{pre}$ right away and must work with local copies called val_{new} and $reach_{new}$. The INIT function initializes an empty val_{new} in a way similar to the initialization of val_{pre}. The function ADDREACH updates the reachability list with a new event (as described in Section 3.6.1) and returns the new updated list which is stored in $reach_{new}$.

Then, updated valuations are computed using the valuations rules given in Section 3.6.1 which are implemented by function COMPUTEVAL. To achieve that, the valuations of the monitored subformulas are computed in a depth-inverse fashion (*i.e.,* smaller subformulas first). Valuations of subformula ψ at moment k depend on valuations at moment k and $k-1$ of subformulas which appear in ψ itself. By ordering the computations, we make sure that computed valuations depend only of valuations already computed.

After the valuations have been computed, the satisfaction of Ψ is evaluated with the SAT function. SAT is based on the same valuation rules as COMPUTEVAL but it does not need to compute all valuations for Ψ and its result is not stored. Indeed, a single valuation which satisfy Ψ is enough to return *true* while $false$ is returned if there is no such valuation.

If SAT returns *true* then the event satisfies Ψ and is thus accepted. ACCEPT transmits the information that the event is accepted to the *Event interposition* box of the monitoring architecture. In this case, the event will take place and the memory must be updated accordingly. The UPDATE function accomplishes this by simply storing val_{new} and $reach_{new}$ instead of val_{pre} and $reach_{pre}$ in the *Memory* box.

If SAT returns $false$ then the event does not satisfy Ψ and must be rejected. REFUSE transmits the information to the *Event interposition* box of the monitoring architecture. In this case, the event will not take place and thus memory is not updated.

3.6.3 Complexity Analysis

Let Ψ be a formula to evaluate with regard to model $\mathfrak{M}$ and $\mathcal{D}$ be the domain of discourse of $\mathfrak{M}$. The complexity of the algorithm presented above can be decomposed as follows:

Space

The overall space required by the algorithm consists of the space taken by val_{pre}, val_{new}, $reach_{pre}$ and $reach_{new}$. val_{pre} and $reach_{pre}$ are the contents of the *Memory* box in Figure 3.3 while val_{new} and $reach_{new}$ are temporary spaces initialized upon reception of an event and freed upon decision (see Section 3.6.2). In the following, we simply denote *val* and *reach* and do not distinguish between memory and temporary structures (since they have the same size).

First, we can decompose the size of *val* into the sizes of its elements:

$$space(val) = \sum_{\varphi \in subf_m(\Psi)} space(val[\varphi])$$

At worst, the space taken by the valuations for a given subformula φ (using compact vectors) is all the possible valuations for the free variables of φ:

$$space(val[\varphi]) = |\mathcal{D}|^{free(\varphi)} \leqslant |\mathcal{D}|^{maxfree_m(\Psi)}$$

Where $free(\varphi)$ is the number of free variables in φ and:

$$maxfree_m(\Psi) = max_{\varphi \in subf_m(\Psi)}\ free(\varphi)$$

This gives us the following upper bound overall:

$$space(val) \leqslant \sum_{\varphi \in subf_m(\Psi)} |\mathcal{D}|^{maxfree_m(\Psi)} \leqslant |subf_m(\Psi)| \times |\mathcal{D}|^{maxfree_m(\Psi)}$$

The space taken by *reach* is easier to assess. Indeed *reach* is a function from $\mathcal{D} \times \mathcal{D}$ to $\{0, 1\}$. Thus:

$$space(reach) = O(|\mathcal{D}|^2)$$

Overall, the space complexity of our algorithm is:

$$\boxed{O(|subf(\Psi)| \times |\mathcal{D}|^{maxfree_m(\Psi)} + |\mathcal{D}|^2)}$$

Time

The overall time to process an event depends on ADDREACH, COMPUTEVAL and SAT.

ADDREACH consists in adding a new context (a $O(1)$ operation) to the reachability list of every context (of which there are at most $|\mathcal{D}|$). Thus:

$$time(ADDREACH) = O(|\mathcal{D}|)$$

As we noted in Section 3.6.2, SAT is a simpler problem than COMPUTEVAL. The overall complexity of both functions will thus be dominated by COMPUTEVAL.

COMPUTEVAL(φ) consists in going recursively through the subformulas of φ and applying valuation rules (see Section 3.6.1).

The bottom cases of this recursion are *1)* encountering a valuation which can be computed immediately (*i.e.,* $\approx$, $>$ or $\in$); *2)* encountering a S or Y operator whose valuation has already been computed. It is not possible to encounter a S or Y operator whose valuations have not yet been computed since we have sorted the monitored subformulas so as to avoid this case (see Section 3.6.2). Consequently, over the execution of all the COMPUTEVAL instances (*i.e.,* the entire for loop), the number of valuation rules applied is exactly the number of operators in Ψ which we denote $nbops(\Psi)$.

Even with naive implementations of the set operations, their complexity is in $O((|\mathcal{D}| \times maxs)^2)$ where *maxs* is the size of the largest valuation vector set which appears in the operation. The maximum number of valid valuation compact vectors for subformula φ is $|D|^{free(\phi)}$ since only free variables appear in compact vectors.

Let us denote:

$$maxfree(\Psi) = max_{\varphi \in subf(\Psi)} free(\varphi)$$

where $subf(\Psi)$ is the set of all subformulas of Ψ. Note that $subf$ and thus $maxfree$ are different from $subf_m$ and $maxfree_m$ since they consider all subformulas instead of only those which must be monitored.

With this notation, we can write the following upper bound on the size of a valuation set in Ψ:

$$\forall \varphi \in subf(\Psi),\ |[\![\varphi]\!]^k| < |D|^{maxfree(\Psi)}$$

So the overall complexity of a full for loop of COMPUTEVAL runs is bounded by:

$$\boxed{O(nbops(\Psi) \times |D|^{2\times(maxfree(\Psi)+1)})}$$

Which dominates the complexity of ADDREACH and is thus the overall time complexity of our algorithm.

Discussion

Neither space nor time complexity depend on k which is a very important property. It means that, even in very long executions, monitoring time and memory overheads stay below a bound which depends only on the policy formula and the domain of discourse.

The efficiency of the set operations could very likely be improved and would reduce the exponent in the time complexity.

The exponents which depend on the number of free variables in subformulas cannot easily be improved. Indeed, a large $maxfree$ means that a large number of quantificators are nested which is an intrinsically complex case.

3.7 Evaluation

This section presents an evaluation of our dynamic monitoring algorithm on the Alice-Bob use-case, where the goal is to enforce the isolation policy between Alice and Bob. Thereafter, we discuss specification of security policies.

3.7.1 Isolation Policy

The wanted isolation policy can be described, with IF-PLTL, as follows:

$$\boxed{\Psi = G(\neg(Alice \gg Bob) \wedge \neg(Bob \gg Alice))}$$

Let's suppose the scenario depicted in Figure 3.8 with two domains A (*Alice, WorkerA, AppA*) and B (*Bob, WorkerB, AppB*). Moreover, *AppA* and *AppB* share *File.*

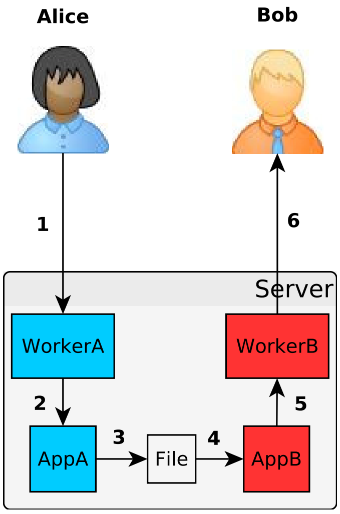


Figure 3.8: Indirect path scenario with Alice and Bob .

The needed sets are configured as follows:

$$\mathcal{D}_{dom} = \{A, B\}$$

$$A = \{Alice, WorkerA, AppA\}$$

$$B = \{Bob, WorkerB, AppB\}$$

$$\mathcal{D}_{ctx} = A \cup B \cup \{File\}$$

Table 3.1 shows each monitoring step of the isolation policy Ψ on the trace represented by Figure 3.8. Upon a new flow event, the *reach* structure is augmented with it. For example, at $k = 2$, the flow $(WorkerA > AppA)$ implies that $AppA$ is reachable from $WorkerA$ but also from $Alice$ due to the presence of $WorkerA$ in $Alice$'s reachability list. However, at $k = 6$, as a consequence of the flow $(WorkerB > Bob)$, Bob becomes reachable from $Alice$. Then, the formula $Alice \gg Bob$ is true leading the policy Ψ to be unsatisfied. Therefore, the event generating the flow $(WorkerB > Bob)$ must be denied and our monitor rollbacks to the previous state $k = 5$. This concludes the enforcement of policy Ψ.

3.7.2 Discussion

As shown previously, the isolation policy is enforced regardless of what event happens; it encompasses many unexpected scenarios other monitors cannot. Let's suppose an other scenario where *File* is a legit configuration file this time; it should not receive any information. Then, the policy can be modified as follows:

$$\Psi_2 = \Psi \wedge G((\forall_{ctx} x)\neg(x > File))$$

With policy Ψ_2, any attempt to write *File* will be denied but any other indirect path will still be taken into account. In our opinion, it is up to the security officer to specify a policy in accordance with the system's expected behavior.

We have presented our logic as an “ideal” security mechanism. But one of our goal is to have a correspondence between a configuration of a protection mechanism such as SELinux and a formula in IF-PLTL. Though configurations are out of the scope of this Thesis as they are achieved by project partners, we want to show the intuition in Table 3.2.

3.8 Conclusion

In this chapter, we have hightlighted the lack of suitable formalism of security policies for distributed systems. Indeed, we need to encompass direct and indirect information flows while having a mechanism agnostic language. Therefore, we have proposed IF-PLTL (Information-Flow Past Linear Time Logic). We have shown that information flow cannot be captured directly but they are obtained from low-level events. Accordingly, in Section 3.4, we have detailed how to construct Information Flow Traces from Observable Events Traces. Then, we have described IF-PLTL syntax and semantics.

Moreover, we have shown the merit of IF-PLTL and its capacity to express any information flow trace property which are inherently temporal. Our primary goal is to distribute a global property into multiple local properties we know how to enforce using several local mechanims. Using IF-PLTL, we present in Section 4.1 the equivalences enabling this transformation.

Nonetheless, we have presented a dynamic monitor that can enforce any IF-PLTL formula. However, the complexity in time and space of this monitoring algorithm is too high to become popularized in practice.

In the future, we will propose methods to ensure the composability of different properties. We will also work on how IF-PLTL properties can be split to be used by a set of collaborating observers, in particular, with observers at different Cloud layers (IaaS, PaaS and SaaS). An interesting work would be to determine whether an IF-PLTL theory is coherent or not.

k	**Event**	**reach**	$Alice \gg Bob$	$Bob \gg Alice$	$SAT(\Psi)$
1	$(Alice > WorkerA)$	$Alice \rightarrow \{WorkerA\}$	false	false	**true**
2	$(WorkerA > AppA)$	$Alice \rightarrow \{WorkerA, AppA\}$ $WorkerA \rightarrow \{AppA\}$	false	false	**true**
3	$(AppA > File)$	$Alice \rightarrow \{WorkerA, AppA, File\}$ $WorkerA \rightarrow \{AppA, File\}$ $AppA \rightarrow \{File\}$	false	false	**true**
4	$(File > AppB)$	$Alice \rightarrow \{WorkerA, AppA, File, AppB\}$ $WorkerA \rightarrow \{AppA, File, AppB\}$ $AppA \rightarrow \{File, AppB\}$ $File \rightarrow \{AppB\}$	false	false	**true**
5	$(AppB > WorkerB)$	$Alice \rightarrow \{WorkerA, AppA, File, AppB, WorkerB\}$ $WorkerA \rightarrow \{AppA, File, AppB, WorkerB\}$ $AppA \rightarrow \{File, AppB, WorkerB\}$ $File \rightarrow \{AppB, WorkerB\}$ $AppB \rightarrow \{WorkerB\}$	false	false	**true**
6	$(WorkerB > Bob)$	**Alice** $\rightarrow \{WorkerA, AppA, File, AppB, WorkerB,$ **Bob**$\}$ $WorkerA \rightarrow \{AppA, File, AppB, WorkerB, Bob\}$ $AppA \rightarrow \{File, AppB, WorkerB, Bob\}$ $File \rightarrow \{AppB, WorkerB, Bob\}$ $AppB \rightarrow \{WorkerB, Bob\}$	**true**	false	**false**

Table 3.1: Interpretation and satisfaction of the isolation between A and B.

Acces Control	**IF-PLTL**
a read b	$b > a$
a write b	$a > b$
no-read-up	$G((\forall h \in High)(\forall l \in Low)\neg(h > l))$
no-write-down	
confidentiality(S, A)	$(\forall x \in S)(\forall y)(x > y) \rightarrow (y \in S \wedge y \in A)$
integrity(S, A)	$(\forall x \in S)(\forall y)(x < y) \rightarrow (y \in S \wedge y \in A)$

Table 3.2: Correspondence between Access Control and IF-PLTL.

Chapter 4

Security Deployment for Virtualized Distributed Systems

In Chapter 2, we have presented our model of virtualized applications and their security requirements. These requirements are security properties which formal semantics have been detailed in Chapter 3. In this chapter, we present how to deploy virtualized applications on a virtualization-based infrastructure.

Our deployment must consider the enforcement of the application security but also the underlying infrastructure security. Multiple approaches may be envisioned to enforce security properties. We propose two complementary solutions. The first one (described in Section 4.2) is a security-aware placement to enforce security between virtual machines (inter-VM security) which may belong to the same or different applications. Then, the second (described in Section 4.3) is the configuration of security mechanisms to provide security within applications' virtual machines (intra-VM security). But first, we explain how a global property can be preprocessed to be enforced by a set of security mechanisms.

4.1 Preprocessing of Security Requirements

A requirement or property may be specified on a set of entities to simplify the specification step. Typically, a user requires his virtualized application to be isolated from others. This application may be arbitrarily complex but the isolation property protects the whole virtualized application from other tenants. It is much easier for the tenant to state the property relatively to the application rather than to each element of this application. However, a property on an abstract group cannot be directly enforced (*i.e.,* by a single security mechanism), especially if the group combines different types of entities *e.g.,* virtual machines and virtual networks.

In this section, thanks to our formal system which allows for reasoning about (explicit) properties, we first give some equivalences between a property on a set of entities and a set of properties on each individual entity. Then, based on these equivalences, we detail our algorithms to extend an implicit property presented in Section 2.3.2 to explicit properties. Finally, still using the equivalences, we propose **splitting** algorithms which transform a global property into locally enforceable properties.

4.1.1 Equivalence for Confidentiality, Integrity and Isolation

In this subsection, we detail the equivalence relations for the Confidentiality property. The approach (and proof) is strictly the same for Integrity and Isolation properties.

Our confidentiality property states that information may flow from a secured set S to itself or an authorized set A. In our Sam4C modeling language, the property has the signature (in the sense of function signature) $Confidentiality(S, A)$. The corresponding definition in IF-PLTL is:

$$Confidentiality(S, A) \equiv_{def} (\forall_{ctx} x \in S)(\forall_{ctx} y)(x > y) \rightarrow (y \in S \cup A)$$

According to this definition, information may flow from any context x in S to any context y if y is either in the authorized domain A or in the secured domain S. The splitting procedure is valid if there is an equivalence between protecting the confidentiality of the set S and protecting the confidentiality of each entity x in S.

Suppose the set S can be divided into disjoint subsets S_1 and S_2 that is $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \emptyset$, then we have the following equivalence:

$$\boxed{Confidentiality(S, A) \equiv Confidentiality(S_1, A \cup S_2) \wedge Confidentiality(S_2, A \cup S_1)}$$

Proof. In this proof, we simplify the notation $\forall_{ctx}$ by $\forall$ as we do not quantify over other sorts. We start with the definition of $Confidentiality(A, S)$ that is:

$$(\forall x \in S)(\forall y)(x > y) \rightarrow (y \in S \cup A) \tag{P0}$$

We define $S = S_1 \cup S_2$ such as $S_1 \cap S_2 = \emptyset$, then:

$$(\text{P0}) \equiv (\forall x \in S_1 \cup S_2)(\forall y)(x > y) \rightarrow (y \in S_1 \cup S_2 \cup A) \tag{P1}$$

In the previous chapter, we have introduced the simplification:

$$(\forall x \in y)\varphi(x) \equiv (\forall x)\big((x \in y) \rightarrow \varphi(x)\big)$$

Then we obtain:

$$(\text{P1}) \equiv (\forall x)\big((x \in S_1 \cup S_2) \rightarrow \big((\forall y)(x > y) \rightarrow (y \in S_1 \cup S_2 \cup A)\big)\big) \tag{P2}$$

Due to the equivalence between union set and logical disjunction, namely:

$$(x \in S_1 \cup S_2) \equiv (x \in S_1) \vee (x \in S_2)$$

We have:

$$(\text{P2}) \equiv (\forall x)\big(((x \in S_1) \vee (x \in S_2)) \rightarrow \big((\forall y)(x > y) \rightarrow (y \in S_1 \cup S_2 \cup A)\big)\big) \tag{P3}$$

The implication operator → is defined as:

$$p \rightarrow q \equiv \neg p \vee q$$

Accordingly, we have:

$$(\text{P3}) \equiv (\forall x)\big((\neg(x \in S_1) \wedge \neg(x \in S_2)) \vee \big((\forall y)(x > y) \rightarrow (y \in S_1 \cup S_2 \cup A)\big)\big) \tag{P4}$$

Because of the distributivity of the logical operators $\wedge$ and $\vee$ that is:

$$(p \wedge q) \vee r \equiv (p \vee r) \wedge (q \vee r)$$

We obtain:

$$(\text{P4}) \equiv (\forall x) \begin{cases} \big(\neg(x \in S_1) \vee ((\forall y)(x > y) \rightarrow (y \in S_1 \cup S_2 \cup A))\big) \\ \wedge \\ \big(\neg(x \in S_2) \vee ((\forall y)(x > y) \rightarrow (y \in S_1 \cup S_2 \cup A))\big) \end{cases} \tag{P5}$$

We use the previous implication operator equivalence:

$$(\text{P5}) \equiv (\forall x) \begin{cases} \big((x \in S_1) \rightarrow ((\forall y)(x > y) \rightarrow (y \in S_1 \cup S_2 \cup A))\big) \\ \wedge \\ \big((x \in S_2) \rightarrow ((\forall y)(x > y) \rightarrow (y \in S_1 \cup S_2 \cup A))\big) \end{cases} \tag{P6}$$

Finally, the universal quantifier distributes over conjunction, that is:

$$\forall x(\varphi(x) \wedge \psi(x)) \equiv (\forall x \varphi(x)) \wedge (\forall x \psi(x))$$

It gives us the result:

$$(\text{P6}) \equiv \begin{cases} (\forall x)\big((x \in S_1) \rightarrow ((\forall y)(x > y) \rightarrow (y \in S_1 \cup S_2 \cup A))\big) \\ \wedge \\ (\forall x)\big((x \in S_2) \rightarrow ((\forall y)(x > y) \rightarrow (y \in S_1 \cup S_2 \cup A))\big) \end{cases} \tag{P7}$$

Which concludes our equivalence:

$$(\text{P7}) \equiv Confidentiality(S_1, A \cup S_2) \wedge Confidentiality(S_2, A \cup S_1)$$

□

For the Integrity and Isolation, the same proof (hence split) hold providing the IF-PLTL definitions given afterwards.

Our integrity property states that information may flow to a secured set S from itself or an authorized set A, that is:

$$Integrity(S, A) \equiv_{def} (\forall_{ctx} x \in S)(\forall_{ctx} y)(x > y) \rightarrow (y \in S \cup A)$$

Our isolation property is the conjunction of confidentiality and integrity, that is:

$$Isolation(S, A) \equiv_{def} Confidentiality(S, A) \wedge Integrity(S, A)$$

Therefore, we obtain the following equivalences:

$$\boxed{Integrity(S, A) \equiv Integrity(S_1, A \cup S_2) \wedge Integrity(S_2, A \cup S_1)}$$

$$\boxed{Isolation(S, A) \equiv Isolation(S_1, A \cup S_2) \wedge Isolation(S_2, A \cup S_1)}$$

In summary, we have presented some equivalences for Confidentiality, Integrity and Isolation properties stated with direct flows. For more complex properties *e.g.,* with indirect flows or temporal modalities, new specific yet not trivial equivalences must be devised. In the next following subsections, we use these equivalences in two ways. First, we present how to split an *implicit property* which is easy to specify for a user into *explicit properties* we must enforce. Then, we detail how to split an *explicit property* into locally enforceable properties *i.e.,* that we know some security mechanisms can enforce directly and locally on a VM or host.

4.1.2 Implicit to Explicit Properties

As previously said, our Confidentiality, Integrity and Isolation properties have two parameters, the secured set and the authorized set. In Section 2.3.2, we have discussed the possibility of specifying *implicit properties* such as the user does not express the authorized set but only the secured set. In this subsection, we show how an *implicit property* can be extended into an *explicit property.*

We have introduced in Chapter 2 the **Security Domain** as an abstract group. Figure 4.1 depicts a security domain DOM comprising 3 VMs *i.e.,* VM1 to VM3 and one VNet (Virtual Network) INTRANET.

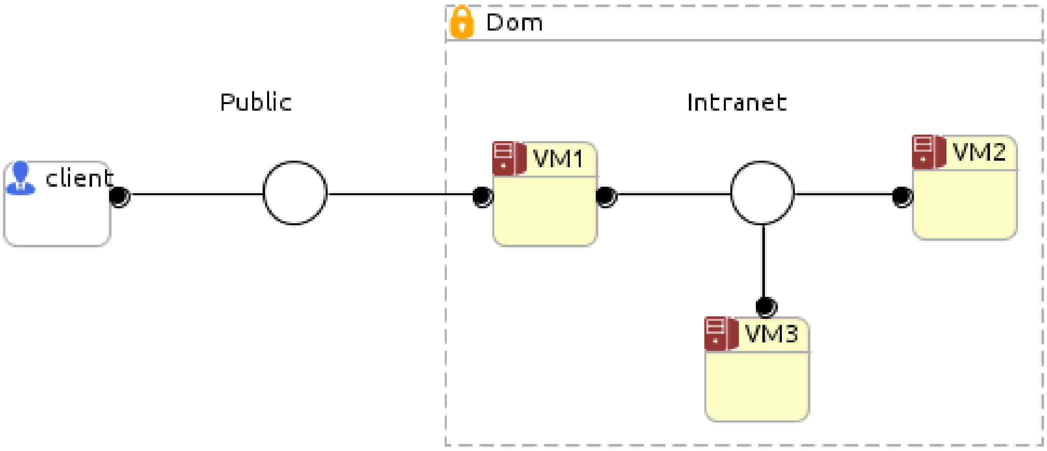


Figure 4.1: Sam4C Model with a security domain of 3 VMs and 2 VNets.

Considering the implicit property below, the problem is to determine the corresponding set of explicit properties.

```
#property Isolation(Dom);
```

Informally, the virtualized application model (see Figure 4.1) shows that the three VMs (*i.e.,* VM1, VM2 and VM3) should be able to exchange information through INTRANET and only VM1 should also communicate with PUBLIC meaning *the authorized set will be different for both cases.* These characteristics must be preserved when extending the implicit isolation property.

To do so, we consider the model as a *graph* where VMs and VNets are *nodes* and VLink are *edges*. The security domain *Dom* is abstract *i.e.,* it is an alias for the set $\{VM1, VM2, VM3, Intranet\}$. Then, the equivalent property is:

```
#property Isolation({VM1, VM2, VM3, Intranet});
```

We distinguish two secured sets: the **border set** and the **internal set**.

Definition 16 (Border Set)**.** *Given a set S of nodes, the border set of S includes all nodes n_i such as it exists at least one edge between n_i and a node not in S.*

Definition 17 (Internal Set)**.** *Given a set S of nodes, the internal set of S includes all nodes n_i such as it does not exist any edge between n_i and a node not in S.*

In Figure 4.1, VM1 is in the border set of DOM, and the others (*i.e.,* VM2, VM3 and INTRANET) is in the internal set of DOM.

The virtualized application model is a graph $G = (V, E)$ where G is a pair (V, E) with V the set of nodes and E the set of edges. We formalize the model's constraints as a function $Auth(x, S, G)$ with x a node, S a set of nodes (including x) and G the model graph. *Auth* is defined as follows:

$$Auth(x, S, (V, E)) = \begin{cases} S \textbf{ if } (\forall x_i \in V)(x, x_i) \in E \rightarrow x_i \in S \\ S \cup \{x_i \mid (\forall x_i \in V)(x, xi) \in E\} \textbf{ otherwise} \end{cases}$$

We can see that *Auth* returns S if x is in the internal set and the list of neighbors union S if x is in the border set. As a result, *Auth* returns the same set (*i.e.,* S) for any internal entities and specific sets for any border entities. Then, using our property equivalences, we define S_1 as the border set S_{border} and S_2 as the internal set $S_{internal}$. The conditions $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \emptyset$ both hold. Thus, we have (*Prop* stands for Confidentiality, Integrity or Isolation):

$$ImplicitProp(S) \equiv \begin{cases} (\forall x_i \in S_{internal})(Prop(\{x_i\}, Auth(x_i) \cup S_{internal} \cup S_{border} - \{x_i\})) \\ \wedge \\ (\forall x_b \in S_{border})(Prop(\{x_b\}, Auth(x_b) \cup S_{internal} \cup S_{border} - \{x_b\})) \end{cases}$$

Because $Auth(x_i) = S$ for all $x_i \in S_{internal}$, we have:

$$\begin{aligned} & (\forall x_i \in S_{internal})(Prop(\{x_i\}, Auth(x_i) \cup S_{internal} \cup S_{border} - \{x_i\})) \\ \equiv\ & (\forall x_i \in S_{internal})(Prop(\{x_i\}, S \cup S_{internal} \cup S_{border} - \{x_i\})) \\ \equiv\ & (\forall x_i \in S_{internal})(Prop(\{x_i\}, S - \{x_i\})) \\ \equiv\ & Prop(S_{internal}, S - S_{internal}) \\ \equiv\ & Prop(S_{internal}, S_{border}) \end{aligned}$$

Therefore, we finally obtain:

$$ImplicitProp(S) \equiv \begin{cases} Prop(S_{internal}, S_{border}) \\ \wedge \\ (\forall x_b \in S_{border})(Prop(\{x_b\}, Auth(x_b) \cup S_{internal} \cup S_{border} - \{x_b\})) \end{cases}$$

This equation is used in Algorithm 4.1 to compute the set of explicit properties from implicit properties. In this algorithm, an implicit property p_i is tuple (T, S, g) where T is the type of property (*i.e.,* Confidentiality, Integrity or Isolation), S the secured set and g the grade. The grade is the quality of protection as discussed in Section 2.3.2. It can be viewed as an annotation that is preserved during the procedure.

Algorithm 4.1 Implicit to Explicit Procedure

```
Input: $P_{implicit}$: set of implicit properties to extend.
Variables: $P \leftarrow \varnothing$
  for all $p_i = (T, S, g) \in P_{implicit}$ do
    $S_{border} \leftarrow \text{listBorder}(S)$
    $S_{internal} \leftarrow S - S_{border}$
    $P \leftarrow P \cup (T, S_{internal}, S_{border}, g)$
    for all $x \in S_{border}$ do
      $A_{neighbors} \leftarrow \text{listNeighbors}(x)$
      $P \leftarrow P \cup (T, \{x\}, S_{internal} \cup A_{neighbors} \cup S_{border} - \{x\}, g)$
  return $P$
```

From our example depicted Figure 4.1, the result of Algorithm 4.1 is:

```
#property Isolation({VM2, VM3, Intranet},{VM1});
#property Isolation({VM1},{Public, VM2, Intranet, VM3});
```

4.1.3 Model-based Property Split

We have presented how to extend an implicit property to an explicit one. Now, the problem is to determine the result of splitting an explicit property into individual properties. To do so, we make use of our previous equivalences in two ways. Firstly, we show how to split a global property *i.e.,* a property with a secured set of entities, to local properties *i.e.,* properties on individual secured entities. Secondly, the previous local properties are specified on multiples types of entities *e.g.,* VMs and VNets but most mechanisms are dedicated to a single type *e.g.,* either to protect VMs or to protect networks. Therefore, we show how to discriminate different types of entities and split a property into **typed properties** *i.e.,* properties where the secured set only contains entities of the same type.

Local properties split

A local property is a property with an individual secured entity. Therefore, if we denote e an individual entity in the secured set S then we can instantiate $S = S_1 \cup S_2$ with $S_1 = \{e\}$ and $S_2 = S - \{e\}$. Accordingly, we obtain the following equivalence:

$$Prop(S, A) \equiv Prop(\{e\}, A \cup (S - \{e\})) \wedge Prop((S - \{e\}), A \cup \{e\})$$

Let a property P be a tuple $(Prop, S, A, g)$ with $Prop$ the type of property (*e.g.,* Confidentiality), S the secured set, A the authorized set and g the grade. We include the grade (as introduced in 2.3.2) to demonstrate that it is preserved through the procedure, meaning any global property with a required quality of protection (*i.e.,* the grade) is split into local property with exactly the same quality. Algorithm 4.2 splits a global property into *singleton* properties *i.e.,* with a unique element in the secured set.

Algorithm 4.2 Singleton Split Procedure

```
Variables: $P = (Prop, S, A, g)$
  if $|S| = 1$ then
    return $\{P\}$
  $PSet = \varnothing$
  for all $x \in S$ do
    $PSet = PSet \cup (Prop, \{x\}, A \cup S - \{x\}, g)$
  return $PSet$
```

Let us apply Algorithm 4.2 to the scenario in Figure 4.1. If we consider the following (explicit) property in input:

```
#property Isolation({VM2, VM3, Intranet},{VM1});
```

Then, the result of this singleton splitting procedure is:

```
#property Isolation({VM2},{Intranet, VM3, VM1});
#property Isolation({VM3},{Intranet, VM1, VM2});
#property Isolation({Intranet},{VM3, VM1, VM2});
```

Typed properties split

In the result given above, the *singleton properties* contain entities of different types in the authorized set *e.g.,* Intranet, VM1. Typically, a security mechanism is dedicated to protect a single type of entity: different mechanisms are used to protect a VNet or a VM. Accordingly, we propose a second procedure to split our properties into **typed properties** that is properties with a single type.

First, we must consider all existing types. In a **Security Domain**, only two types exist namely VM and VNet. Then, the IF-PLTL definitions of Confidentiality, Integrity and Isolation are adapted to reflect the types, we obtain:

$$Confidentiality_{type}(S, A) \equiv_{def} (\forall x \in S)(\forall y \in type)(x > y) \rightarrow (y \in S \cup A)$$

$$Integrity_{type}(S, A) \equiv_{def} (\forall x \in S)(\forall y \in type)(x < y) \rightarrow (y \in S \cup A)$$

$$Isolation_{type}(S, A) \equiv_{def} Confidentiality_{type}(S, A) \wedge Integrity_{type}(S, A)$$

We consider that information may flow between a combination of VNet or VM *e.g.,* VNet to VM or VM to VM. Providing the authorized set is divided into VMs and VNets (*i.e.,* $\forall y$ range only on VMs and VNets), we can easily prove the equivalence:

$$Prop(S, A_{vm} \cup A_{vnet}) \equiv Prop_{VM}(S, A_{vm}) \wedge Prop_{VNet}(S, A_{vnet})$$

Algorithm 4.3 Typed Split Procedure

```
Variables: $P = (Prop, S, A, g)$
  $PSet_{typed} = \emptyset$
  for all $type \in \{vm, vnet\}$ do
    $A_{typed} = \{x \mid x \in A \wedge x \in type\}$
    $PSet_{typed} = PSet_{typed} \cup (Prop_{type}, S, A_{typed}, g)$
  return $PSet_{typed}$
```

Let us apply Algorithm 4.3 to the scenario in Figure 4.1. If we consider our previous singleton properties in input:

```
#property Isolation({VM1}, {Public, Intranet, VM2, VM3});
#property Isolation({VM2},{Intranet, VM3, VM1});
#property Isolation({VM3},{Intranet, VM1, VM2});
#property Isolation({Intranet},{VM3, VM1, VM2});
```

Then, the result of this typed splitting procedure is:

```
#property IsolationVM({VM1}, {VM2, VM3});
#property IsolationVNet({VM1}, {Public, Intranet});

#property IsolationVM({VM2},{VM3, VM1});
#property IsolationVNet({VM2},{Intranet});

#property IsolationVM({VM3},{VM1, VM2});
#property IsolationVNet({VM3},{Intranet});

#property IsolationVM({Intranet},{VM3, VM1, VM2});
#property IsolationVNet({Intranet});
```

Consequently, given these local typed properties, we must find a mechanism to enforce each one. For example, we show in Section 4.2 how to enforce the isolation of VM1 from any other VMs except for VM2 and VM3 using a placement-based strategy. Similarly, we could enforce the isolation of VM1 from any VNet except for Public and Intranet by simply configuring only two network interfaces in VM1.

4.1.4 Conclusion

In resume, we have first detailed how to split an *implicit property* into *explicit properties.* Then, we have shown how to split a global (explicit) Confidentiality, Integrity or Isolation property on VMs and VNets into their local typed counterparts. In future work, this approach could be extended to other kinds of properties or for properties on entities *within* virtual machines *e.g.,* Data, Service. In Section 4.2, we present our solution to enforce the isolation between VMs only. Then in Section 4.3, we detail our generic solution based on the configuration of security mechanisms for other properties in particular those protecting entities within VMs. Enforcing properties on virtual networks is part of our future work though we discuss a potential solution in the conclusion of this chapter.

4.2 Placement-based Security

In virtualized distributed systems such as Clouds, multitenancy and shared resources facilitate attacks and thefts by altering the level of isolation between tenants, processes or virtual machines. While virtualization could be seen as an isolation mechanism between tenants, we show in this section that it is not the case [103]. Accordingly, it is required to have another mechanism to enforce the isolation.

Furthermore, the security of a system is as strong as the weakest link. And, even with a perfect information flow (or access) control mechanism at the system level (*e.g.,* Hypervisor or Operating system), information can be silently leaked out (or accessed) by exploiting (unwillingly) unsecured design or implementation; this is called *covert channels.* The literature exhibits multiple covert channels attacks that have been successfully conducted in Cloud environments [103, 123, 124] (*e.g.,* Amazon EC2) thanks to the shared hardware components between an attacker and a victim. Previous works opt for different approaches to tackle this issue: detection of attacks, fine-grained tracking of resources usage and placement algorithms. Placement algorithms under collocation/anti-collocation constraints specify if 2 VMs can share the same Physical Machine (PM) or not. Accordingly, they do not take into account any scenarios where sharing resources is reasonable when measuring information leakage.

In this section, we propose a new resource allocation mechanism under information leakage constraints. Our mechanism takes into account microarchitectural components as they are one of the main reasons of covert channels. Moreover, it allows to have a finer-grained allocation than previous approaches and thus reduces the quantity of resources wasted due to security constraints. But first, we need a way to present the quantity of information that can be leaked between 2 applications. While our allocation algorithm could use any covert-channel metric, due to the lack of proper metric, we propose a new information leakage metric that can be used by a tenant and is both application and hardware independent. This metric quantifies information leakage through microarchitectural covert channels based on the achievable bitrate between 2 applications [78, 115].

Using this metric, a tenant can express its isolation properties and specify the acceptable leakage that fits his security needs. For example, banks, governments and medical establishments usually require more secure setups than research institutes running experiments. Using the virtualized application model and our metric, we propose our new resource allocation mechanism. Furthermore, we present how we have tackled the specificity of NUMA (Non-Uniform Memory Access) allocation policies to encompass both traditional and modern architectures, and proposed a model for our algorithm.

4.2.1 State of the Art

The major problem of virtualization is a weak isolation. Ristenpart *et al.* [103] discuss multiple approaches to exploit the co-residency in Cloud environments such as Cross-VM performance degradation, Denial of Service (DoS) attacks or stealing cryptographic keys, thus demonstrating the reality of security threats. They have shown the feasibility of collocating their VMs on the same physical machine as business targets in EC2. This work demonstrates that virtualization is far from being sufficient to have a secure environment.

In this section, we focus on the *isolation problem* of VMs sharing the same hardware and hypervisor. Furthermore, according to our previous definition of isolation, we will not consider performance interference. Indeed, it does not allow information modification or leak even if it is due to improper isolation [116].

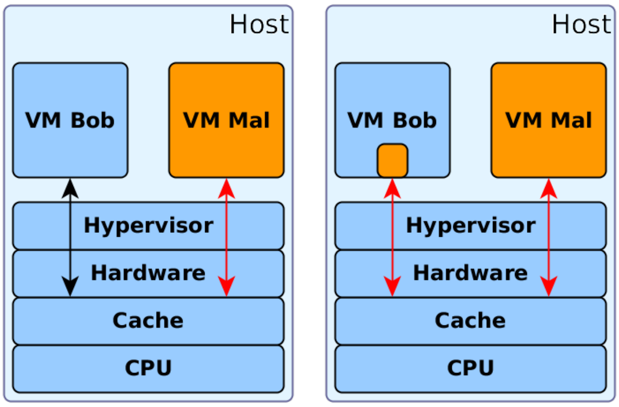
Microarchitectural Timing Covert Channels

Cloud providers can provide a large range of security mechanisms to prevent unauthorized information flows. But, despite all the effort, there is still a potential risk of data leakage in the Cloud, that is **covert channels**.

A covert channel is an attack which bypasses the control mechanism using legal means to leak information to unauthorized neighbors. A covert channel breaks the confidentiality property and thus, the isolation property. Therefore, even a perfect control mechanism is useless against covert channels.

Covert channels should not be confused with **side channels**. A covert channel exists when 2 cooperative entities, let say TROJAN and SPY, use a common protocol to communicate or exfiltrate information. For a side channel, only one process, the Spy, collects unwillingly disclosed information. Because the leaking process can be considered as an unintentional Trojan, we argue that side channels are special cases of covert channels. Recent exploits (*e.g.,* Heartblead, Ghost) have shown the easiness of remotely uploading and executing a malicious code. Side and covert channels may exist at any layer *i.e.,* hardware and software. Our approach is dedicated to improve the isolation between VMs. Therefore, we only consider covert channels harmful to VMs *i.e.,* microarchitecture-based covert channels.

As depicted Figure 4.2, side and covert channels may exploit microarchitectural components like caches or CPUs to infer or convey information. In this illustration, two VMs Bob and Mal are collocated on the same host *i.e.,* physical machine. In Figure 4.2a, VM Mal “listens” to the microarchitecture to extract information from VM Bob, this is a side channel. Whereas in Figure 4.2b, VM Mal receives information from a TROJAN represented by a square in VM Bob.



(a) Side Channel (b) Covert Channel

Figure 4.2: Microarchitectural Side and Covert Channels

Covert channels are categorized in **covert storage channels** and **covert timing channels**.

A covert storage channel exploits a standard data channel to encode secret information. This problem has been described in 1983 by G. Simmons [110] as the prisoner's problem. Two prisoners, Alice and Bob, are detained in distinct cells. Their goal is to elaborate an escape plan. They can communicate with each other by letters only, conveyed by the warden. But, they have to consider that the warden is free to read the content of the letters, remove (part of) it or even forge new content. Consequently, Alice and Bob have to encode their discussion in an apparently carefree text. This attack is popular for network protocols, for example by using the reserved or unused bits of a frame to convey information. As covert storage channels are not due to hardware components but software design, we will focus mainly on *covert timing channels*.

A covert timing channel exploits access timings of shared resources. In his work [97], Percival took advantage of cache hit and miss to convey "0" and "1" respectively. With collocated VMs sharing multiple resources (hardware and system), virtualization-based environments are conducive for such exploits. Building a covert timing channel in a public commercial Cloud (such as Amazon EC2) has been proven feasible [103, 123, 124]. A reliable bandwidth of just a hundred bits per second is enough to silently extract hundreds of 1024-bytes private keys, or tens of thousands of credit cards in a day. The DoD guideline [115] (TCSEC / Orange Book) characterizes a covert channel by its bitrate and error rate. It suggests that covert channels exceeding a threshold of 0.1 bit per second should be audited and be of concern to security if it performs over 1 bit per second.

In a virtualized environment, covert timing channels can be conducted using some hardware or system components. In Table 4.1, we draw a summary of previous covert timing channels works. For the same component, reported bitrates can differ up to 6 orders of magnitude (*e.g.*, 0.2 bps and 190.4 kbps for L2 cache covert channel) depending on the experimental environment (*i.e.*, setup column). This gap cannot be explained by hardware differences alone (*e.g.*, CPU clock speed, cache size). How the attack is implemented also has a major impact on the bitrate.

Table 4.1: Covert timing channels summary for Cloud environments

Paper	Component	Bitrate	Error Rate	Setup	Access Pattern	Error Correction
[97]	L1	3.2 Mbps	<25%	Lab	Parallel (SMT)	No
	L2	800 kbps	<25%	Lab	Parallel (SMT)	No
[120]	SMT/FU	500 kbps	NC	Lab	Parallel (SMT)	NC
	Speculation	200 kbps	NA	Lab	Sequential	Noiseless
[124]	L2	262.47 bps	NA	Lab	Sequential	Noiseless
	L2	3.75 bps	8.59%	EC2	Sequential	DF
[123]	Memory bus	746.8 bps	0.09%	Lab	Parallel	FEC
	Memory bus	107.9 bps	0.75%	EC2	Parallel	FEC
	L2	190.4 kbps	NC	Lab	Parallel	NC
[93]	CPU	0.49 bps	NA	Lab	Sequential	Noiseless
[122]	Xen Shared-Mem	174.98 bps	2%	Lab	Sequential	No
[103]	Memory bus	0.006 bps	NC	EC2	NC	NC
	Hard disk	0.0005 bps	NC	EC2	NC	NC
	L2	0.2 bps	NC	EC2	Sequential	DF

NA: do Not Apply; NC: Not Communicated; FEC: Forward Error Correction; DF: Differential Coding.

Mitigation techniques

Covert channels in multitenant environments pose a real threat with bitrates largely over the 1 bit per second standard threshold. A covert channel is by definition a reliable data channel, therefore reducing the bandwidth, preventing the channel from being reliable (*i.e.,* increasing the error rate) or removing the channel are the basic ideas for mitigation techniques.

We discuss below the mitigation possibilities given 3 perspectives: the tenant, the provider and the hardware manufacturer.

Tenant The tenant has limited possibilities against covert channels. HomeAlone [131] is a detection approach to detect unusual L2 cache usage without relying on hardware support nor hypervisor modification. But HomeAlone's oracle is not perfect and the Spy can try to evade the detection. Moreover, according to Wu *et al.* [123], for memory bus covert timing channels, this approach would be subject to high performance overhead and high false positive rate due to non-determinism and higher access latencies. To sum up, a detection approach at the tenant level cannot be generalized and do not apply to yet to discover covert timing channels.

Provider On the other side, the Cloud provider has more latitude to mitigate covert channels. He can use detection techniques at the hypervisor level with lower overhead. Moreover, he can modify the scheduling policy to implement cache partitioning or page coloring to isolate cache/memory accesses [100, 114] at the cost of performance. Others existing preventive approaches are dedicated instances or collocation/anti-collocation placement. In this case, the isolation properties can be modeled as collocation and anti-collocation constraints [31, 65]. Nevertheless, they require to express the list of tenants with whom each VM is allowed to share resources (or not). Moreover, such measures are costly for the tenant as he pays extra charges for dedicated physical machines as these approaches use coarse grain resource allocation model.

On modern public Clouds such as Amazon EC2 and Microsoft Azure, hardware multithreading called simultaneous multithreading (SMT) is disabled as L1 cache-based covert channels attacks are easy to build. For example, L1 caches that are dedicated to one core are the easiest way to create covert channels between VMs [97]. The NoHype concept [70] consists in removing the virtualization layer while retaining the key features enabled by virtualization. They limit covert channels by enabling one VM per core but as we have shown, other covert channels exist within the microarchitecture components.

Hardware manufacturer A covert channel is based on a faulty implementation. Thus, the hardware design is the initial reason of most covert channels. The solution would be to integrate security concerns when designing hardware components. Needless to say, it will not fix the issues on hardware already in production. Nevertheless, hardware design is out of scope.

Summary Because of the lack of near-future improvement in hardware design and the specificity of detection techniques, we propose a covert channel aware placement solution. Outsourced applications do not have the same level of criticality. For example, a private individual's website is less threatened by covert channels than a banking or government application. Therefore, we consider the tenant to be responsible for specifying an acceptable information leakage risk and then, the Cloud provider has automated procedures to decide whether the tenant's VM should share L1/L2/L3 caches, memory bus, etc. with other VMs. Such solution can be applied to any Cloud and easily enriched with newly discovered covert channel attacks. The placement algorithm has to use a general metric to deduce a value from a given placement/hardware configuration and compare it with the tenant's risk requirement but also to compare two configurations and deduce the more secure one. The following part discusses existing metrics and devises a new metric to tackle practicability issues.

Because of the lack of near-future improvement in hardware design and the specificity of detection techniques, we propose a covert channel aware placement solution. Outsourced applications do not have the same level of criticality. For example, a private individual's website is less threatened by covert channels than a banking or government application. Therefore, we consider the tenant to be responsible for specifying an acceptable information leakage risk and then, the Cloud provider has automated procedures to decide whether the tenant's VM should share L1/L2/L3 caches, memory bus, etc. with other VMs. Such solution can be applied to any Cloud and easily enriched with newly discovered covert channel attacks. The placement algorithm has to use a general metric to deduce a value from a given placement/hardware configuration and compare it with the tenant's risk requirement but also to compare two configurations and deduce the more secure one. The following part discusses existing metrics and devises a new metric to tackle practicability issues.

Covert channel aware metric

Due to the recentness of Cloud (and virtualization) covert channel works, the literature is quite poor in covert channel metric propositions. Published in 2012 and after, the reference works are the metrics proposed by Demme J. *et al.* [40] and Zhang T. *et al.* [129] with respectively the Side channel Vulnerability Factor (SVF) and the Cache Side channel Vulnerability (CSV). Firstly, both metrics are design for side channels and not covert channels, but because side channels and covert channels are intrinsically related, theses metrics cannot be discard based on this sole argument to motive our new information leakage metric.

As illustrated Figure 4.3, SVF and CSV are float values between 0 and 1 reflecting the degree of information leakage an observer can see from an execution. They correlate the oracle execution traces with the leaked execution traces viewed by another process in terms of cache accesses, CPU loads, etc. The approach is thus hardware-independent but the obtained value is application-specific. Indeed, the evaluation is based on the execution of a cryptographic library and it is hard to deduce if a given value would be roughly the same for an arbitrary application.

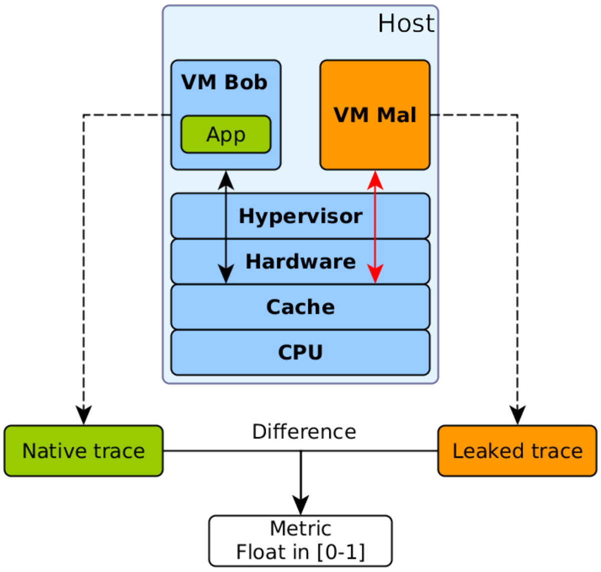


Figure 4.3: SVF and CSV workflow

One problematic is to allow a tenant to specify a value as an acceptable risk. Because SVF and CSV are correlation factors, they can be easily used for comparison but except for limit values (0 and 1) it is quite a challenge to give a practical meaning to an intermediate value (*e.g.,* 0.467), even for a knowledgeable tenant.

Therefore, both SVF and CSV are unspecifiable in practice, and thus we must come up with an alternative *covert channel aware metric* presented thereafter.

4.2.2 Information Leakage Quantitative Metric

We have previously shown that both SVF and CSV metrics are not satisfying. Because the DoD guideline [115] (TCSEC / Orange Book) indicates that a covert channel (characterized by its bitrate and error rate) over 1 bit per second (bps) should be of concern to security, we use it as a reference value for our metric.

Therefore, we define the Information Leakage Metric as follows:

Definition 18 (Information Leakage Metric)**.** *Ratio between the covert channel theoretical bitrate in a noiseless environment and the reference value (1 bps).*

Unlike SVF and CSV, our metric characterizes the *worst case scenario*. In a security context, such scenario corresponds to the *maximal bitrate* a covert channel can theoretically achieve. Thus, to compute this value, we consider *a noiseless environment* and *a parallel setup* (the Trojan and the Spy run on distinct cores). Because our Information Leakage approach is based on a core characteristic (*i.e.,* the bitrate), a value exists for any timing covert channel though it can be complex to compute it.

In the following, we exemplify the approach with a cache-based timing covert channel.

Cache-based Timing Covert channels bitrate in Virtualized Environment

In this part, we detail how a cache-based timing covert channels works using Percival *et al.* technique [97].

The idea is to time the latency of accessing memory addresses. We distinguish two cases, that is, if the accessed data are already in the cache, then the latency is small; else it must be retrieved from the main memory, then the latency is larger. As illustrated in Figure 4.4, the Spy accesses twice a set of cache lines (by accessing memory addresses mapped to it). When the data is in the cache, the latency is under a threshold and so the Spy reads bit “0”, otherwise it reads bit “1”. Its data are now in the cache.

Similarly, to transmit bit “1”, the Trojan accesses the same set of cache lines (by accessing its own memory addresses mapped to it) *i.e.,* he flushes the Spy’s data off the cache. To transmit bit “0”, the Trojan lets the cache in the same state by doing nothing.

Bandwidth estimation for set-associative cache A set-associative cache is divided in sets of cache lines. As shown in Figure 4.5, a memory address can be split into {TAG, SET, OFFSET} bits where OFFSET is the offset in the cache line, SET the set number and TAG the stored value to compare addresses. In a w-way set-associative cache, w addresses mapped to the same cache line can be stored at the same time, generally based on a LRU replacement policy.

As a result, w addresses must be accessed to fully flush one w-way cache line.

Furthermore, virtual to physical address translation rises the problem of addressing uncertainty. Nonetheless, in modern operating systems, the memory space is divided in 4 KB pages and the translation mechanism maps a virtual page to a physical page. Thus, as illustrated in Figure 4.5, the last 12 bits of a memory address remain identical after the translation, leaving *unknownsets* possible cache sets for one virtual memory address where:

$$unknownsets = \frac{nbsets \times linesize}{pagesize} \tag{4.1}$$

Therefore, to evict a Spy’s cache line, the Trojan accesses at least *minlines* addresses where:

$$minlines = w \times unknownsets \tag{4.2}$$

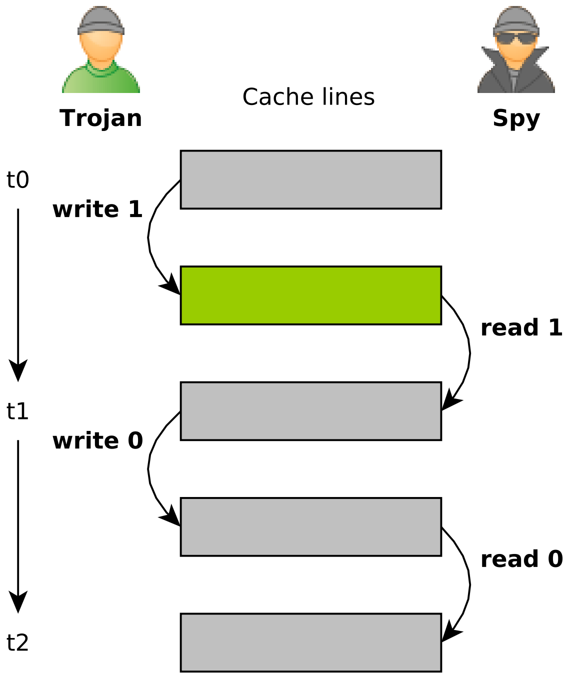


Figure 4.4: Cached-based timing covert channel transmitting "10".

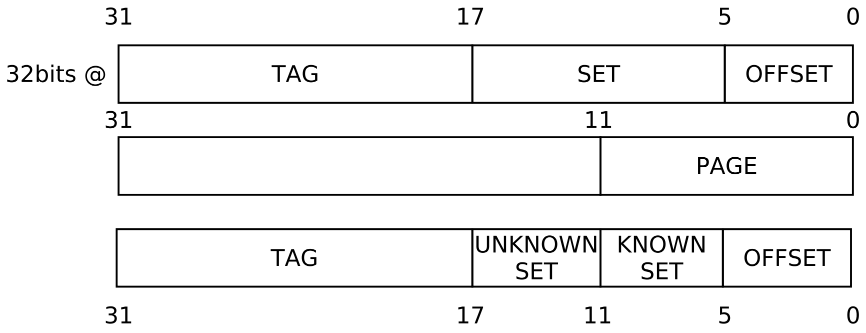


Figure 4.5: Memory address cache line and page mapping.

Given the total cache size is $ttsize = w \times nbsets \times linesize$, to read or transmit a single bit the trojan accesses at least:

$$minlines = \frac{ttsize}{pagesize} \tag{4.3}$$

Finally, we suppose the same probability to transmit bit "0" or bit "1" *i.e.*, $P(0) = P(1) = 1/2$. Thus, the final time to read or write a bit is:

$$\boxed{\frac{(latency_{cached} + latency_{flushed})}{2} \times \frac{ttsize}{pagesize} \tag{4.4}}$$

Experiment To show the feasibility of building a timing covert channel, we have conducted a proof-of-concept experiment which result is depicted Figure 4.6. We have launched two programs running on distinct cores with a shared L2 cache: a writer (or Trojan) and a reader (or Spy). The upper graph presents the time per access in nanoseconds and the lower graph presents the number of L2 cache misses. All values are obtained from the reader. The green line corresponds to the transmission of only bits 1, the red line to transmissions of only bits 0 and the blue line is an alternation between the two. We have a clear correlation between L2 cache misses and access times suggesting the influence of the former on the latter. In this coarse-grain experiment, the bandwidth is 100 bits per second which is far from the theoretical maximum bitrate of 9.551 Kbps.

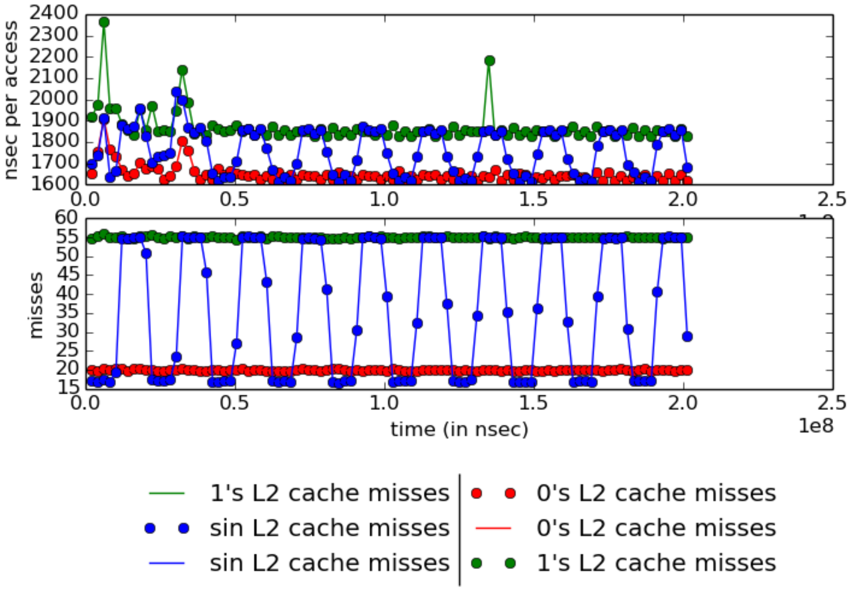


Figure 4.6: Timing Cached-based Covert Channel on Intel Xeon E5420 QC.

Discussion

Previous works on covert timing channels are mainly based on experimental results. To our knowledge, our work is the first attempt to generalize the computation of cache-based covert timing channel bitrate. For this generalization, we consider the time between writing and reading to be null whereas experimental approaches must implement a *synchronization mechanism*. Indeed, there is a vast range of possibilities for this synchronization between the Trojan and the Spy, and choosing one would alter the quality of our metric. Moreover, this synchronization can be achieved independently from the attack itself. For example, in side channels works, the question is to detect when a cache activity corresponds to a cryptographic computation. If the victim is a web server, a simple solution is to *trigger* it by accessing a SSL-encrypted web page.

Secondly, our information leakage metric can only be based on *known* covert timing channel attacks, thus it encompasses neither unknown existing attacks nor yet-to-discover attack schemes. However, this metric does not rely on execution traces and as so, it applies to any application and, if properly adapted, any hardware. For example, atomic memory operations are improperly emulated in x86 virtualized environment [123]. These atomic operations have been implemented with a big system lock (a lock that freezes all other activities of the system) to keep the memory consistent. Consequently, a memory contention generated by one VM can be seen by another one running on the same physical machine. By observing a contention or its absence, the Trojan and the Spy can transmit respectively bit “1” or “0”.

A maximum theoretical bandwidth is arguably not the best risk metric *i.e.,* a metric accurately reflecting the dangerousness of timing covert channels. Nevertheless, it can apply to any hardware, be automated and we believe it preserves the scale of dangerousness *i.e.,* a low bitrate should mean a low risk when a high bitrate should mean a high risk. To the best of our knowledge it is the only metric with all discussed features. In the next subsection, we shown how we use this risk metric in our placement algorithm to enforce isolation properties between VMs. And then, in Section 4.2.4, we detail our framework to automate the computation of our information leakage metric.

4.2.3 Information Leakage Aware Placement

This work focuses on the placement of VMs with information leakage constraints specified as isolation properties. What we want to demonstrate is the effectiveness of our approach to ensure isolation and what to consider when doing placement-based security *i.e.,* how to enhance state-of-the-art algorithms to take into account these constraints. In this subsection we present how isolation properties between VMs are satisfied (*i.e.,* enforced) during the placement routine. Moreover, we describe the problem of NUMA allocation and show how we tackle this issue. Finally, we prove the NP-completeness of our placement problem.

Isolation Properties Satisfaction

A placement is the association between a VM and a configuration. We define a *configuration* as a set of NUMAs and cores. By *instantiated VMs*, we denote the VMs placed on a physical machine (on a set of NUMAs and cores).

A candidate VM’s configuration is valid if both following conditions are met:

1. All properties of the candidate VM are satisfied regarding instantiated VMs.

2. All properties of instantiated VMs are satisfied regarding the candidate VM.

An isolation property of a first VM vm_1 is satisfied regarding a second VM vm_2 if one of the 2 following conditions is met:

1. vm_2 is allowed to transfer information with vm_1

2. the *information leakage* between their configurations is lower than the one specified by the property (as an acceptable risk).

These two requirements on top of the hardware ones (*i.e.,* CPU, RAM, disk) must be fulfilled to find a suitable configuration. Furthermore, to be able to ensure that all the VMs are respecting their isolation properties (and the ones of the others) during their whole lifetime, it is important to use CPU pinning *i.e.,* to statically associate VMs with cores. Indeed, by not doing so (*i.e.,* if we let a VM changes core during its lifetime or randomly selects one at startup), one of the isolation property could be broken and so the security of the application running inside it as the configuration of the VM would have changed.

Accordingly, our security-aware algorithm must test all configurations for a VM on the physical machines and place it as soon as a configuration is valid. Moreover, the placement routine must update the available resources of the platform. Actually, this update is more complex than it seems due to the way NUMA allocation works. In the following part, we present the NUMA allocation problem and the solution we propose.

The NUMA allocation problem

We must consider real-world microarchitectural allocation schemes to counter real-world microarchitectural attacks (covert channels). Therefore, our proposal is to have a fine-grained control over which (microarchitectural) component is shared among multiple VMs to mitigate potential covert channels. With *libvirt*[1], in addition to the selection of specific cores, we can choose a set of NUMA nodes and one of the 3 available memory allocation policies:

- *interleave* that allocates memory on a given set of NUMA nodes in a round-robin fashion but falls back to other nodes if the allocation is not possible. In the worst case, any NUMA node can be allocated.
- *strict* that only allocates memory on a given set of NUMA nodes (or it fails).
- *preferred* that allocates memory on a given preferred node but falls back to other nodes if the allocation is not possible. In the worst case, any NUMA node can be allocated.

We use the *strict* policy to have a fine-grained control over memory allocation as we have for CPU cores.

Figure 4.7 illustrates how the strict NUMA allocation works. We consider a physical machine composed of 2 NUMA nodes with 2 Gb memory and 2 cores each. A first VM (1 core, 1 Gb memory) is bound to the core c_1 and the closest NUMA node (*i.e.,* the memory banks are directly connected to the core), $Numa_1$. A second VM requiring 3 cores is bound to c_2, c_3, c_4 which are spread on the 2 NUMA nodes. If this VM requires 2 Gb memory, allocating memory from $Numa_2$ is enough. But if 3 Gb memory are required, both $Numa_1$ and $Numa_2$ are chosen.

In terms of microarchitectural attacks, the information leakage between the 2 VMs is simply the bandwidth leakage between their bound cores and NUMAs.

Figure 4.8 illustrates our proposal to deal with the memory segmentation. The first VM (3 cores and 3 Gb of memory) is bound on both $Numa_1$ and $Numa_2$ because it

[1] The API for managing the virtualization layer on Linux: `http://libvirt.org/`

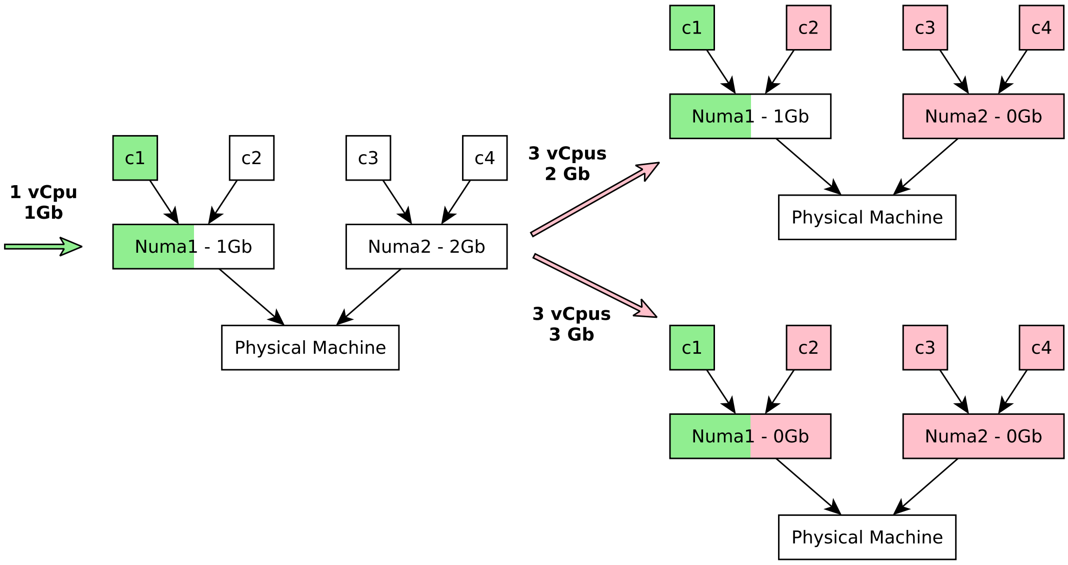


Figure 4.7: Strict memory allocation policy for 2 VMs (1 core then 3 cores).

requires more than 2 Gb. The problem is that there is no indication on how much memory will be left on $Numa_1$ and $Numa_2$, a total of 1 Gb memory being free though. As a result, the second VM requiring 1 core and 1 Gb memory is also bound to $Numa_1$ and $Numa_2$. Our solution is to virtually consider the merging of $Numa_1$ and $Numa_2$, that is $Numa_{1,2}$ which has a total memory of 4 Gb, 1 Gb being free after the placement of the first VM.

Then, in terms of microarchitectural attacks, the information leakage between the 2 VMs is the bandwidth leakage between $\{c_1, c_2, c_3, Numa_1, Numa_2\}$ and $\{c_4, Numa_1, Numa_2\}$.

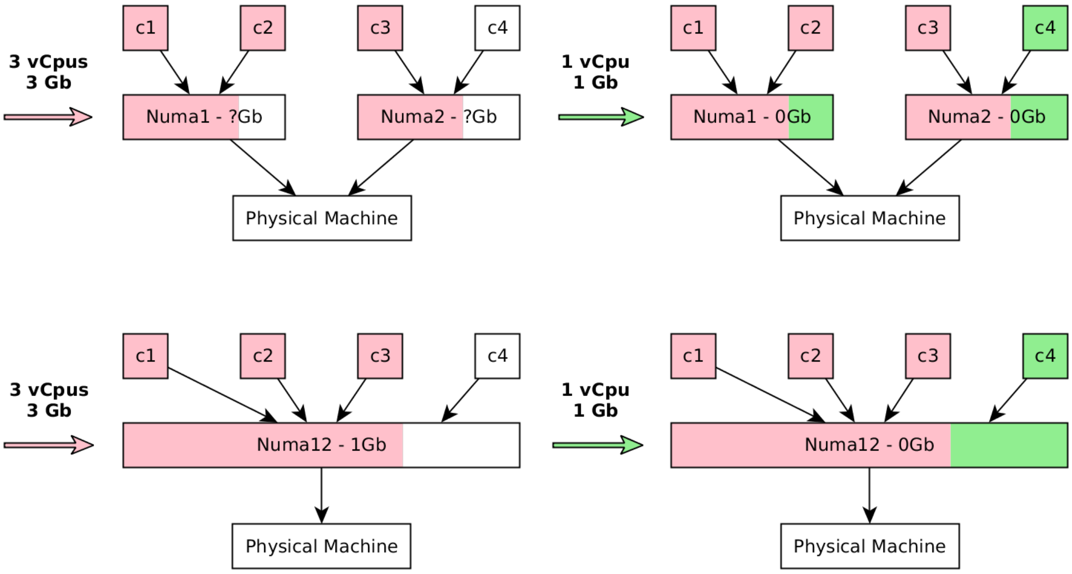


Figure 4.8: Strict memory allocation policy for 2 VMs (3 cores then 1 core).

Metamodel for NUMA allocation Figure 4.9 is an extension of our previous metamodel described in Figure 2.14 (Page 35) to encompass the NUMA allocation scheme.

We call **ComposedNuma** the merging of multiple **VNumas** where a **VNumas** is either a real **Numa** or a **ComposedNuma**. A **VNuma** has a name, a total memory mem_{total}, a free memory mem_{free} and an internal counter of VMs allocated on it vm_{insts}.

Additionally, we have two references: the list of available **Cores** and the list of available **VNumas**. Both refers to components that can be allocated. They consist in respectively:

- The list of **Cores** not executing any VMs,
- the list of **VNuma** that are not part of a **ComposedNuma**.

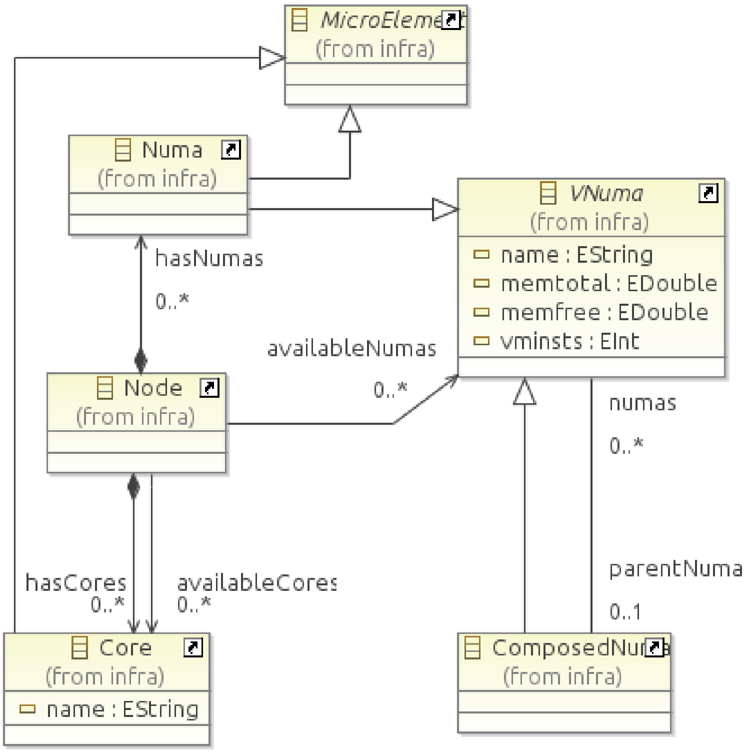


Figure 4.9: Microarchitecture Metamodel with Virtual NUMAs.

NUMA Composition and Decomposition Procedures Our NUMA allocation procedure works as follows. Upon a new memory allocation request, we select a set of NUMAs l_{vnumas} in the list of available NUMAs such as there is enough memory to allocate and call the compose procedure presented in Algorithm 4.4. If l_{vnumas} contains more than 1 NUMA, we merge them and create a new composed NUMA cn, set cn memory properly, and finally we replace l_{vnumas} from the list available NUMAs with cn. Otherwise (l_{vnumas} is a singleton hence the procedure does not apply), we just update *numas* available memory.

Algorithm 4.4 Compose NUMA Procedure

```
Input: $n$: Node, $l_{vnumas}$: List of VNUMAs
Precondition: $|l_{vnumas}| > 1$
Variables: $cn$: ComposedNuma
  $cn.vm_{insts} = 0$, $cn.mem_{free} = 0$, $cn.mem_{total} = 0$
  for all VNuma $vn \in l_{vnumas}$ do
    {Update memory values}
    $cn.mem_{free} += vn.mem_{free}$
    $cn.mem_{total} += vn.mem_{total}$
    {Add sub-vnuma into composed numa and update parent}
    $cn.numas \leftarrow cn.numas \cup \{vn\}$
    $vn.parent \leftarrow cn$
    {Remove sub-vnuma from availability list}
    $n.numas_{available} \leftarrow n.numas_{available} - vn$
  {Add new composed to availability list}
  $n.numas_{available} \leftarrow n.numas_{available} \cup \{cn\}$
  return $cn$
```

In the decomposition procedure detailed in Algorithm 4.5, we delete a composed NUMA if and only if there is no more VM allocated on it. When it is the case, we put the merged NUMAs back in the list of available NUMAs.

Algorithm 4.5 Decompose NUMA Procedure

```
Input: $n$: Node, $vn$: VNuma
  {Decompose numas if possible}
  if $vn$ is not a ComposedNUMA then
    return
  ComposedNuma $cn$ ←ComposedNuma($vn$)
  if $cn.vm_{insts} = 0$ and $cn.parent = \varnothing$ then
    {Remove top-numa}
    $n.numas_{available} \leftarrow n.numas_{available} - cn$
    for all VNuma $vn_{sub} \in cn.numas$ do
      {Remove parent}
      $vn_{sub}.parent \leftarrow \varnothing$
      {Add to availability list}
      $n.numas_{available} \leftarrow n.numas_{available} \cup \{vn_{sub}\}$
      {Recursive decomposition if possible}
      decomposeNuma($n$, $vn_{sub}$)
```

Figure 4.10 depicts the composition phase of our NUMA algorithm with 3 sequentially instantiated VMs. First, vm_1 is allocated onto $Numa_1$ (1 Gb required for 2 Gb available). Then, vm_2 requires 3.5 Gb and we consider $Numa_2$ and $Numa_3$ to be chosen. As a result, a **ComposedNUMA** $Numa_{2,3}$ is created with a total of 4 Gb and 0.5 Gb available after the instantiation of vm_2. At this step, there is 3.5 Gb of memory available spread between $Numa_1$ (1 Gb), $Numa_{2,3}$ (0.5 Gb) and $Numa_4$ (2 Gb). Finally, vm_3 which requires 3.5

Gb is instantiated. A **ComposedNUMA**, $Numa_{1,2,3,4}$ is created for a total of 8 Gb, 3.5 Gb free before the instantiation of vm_3 and none after.

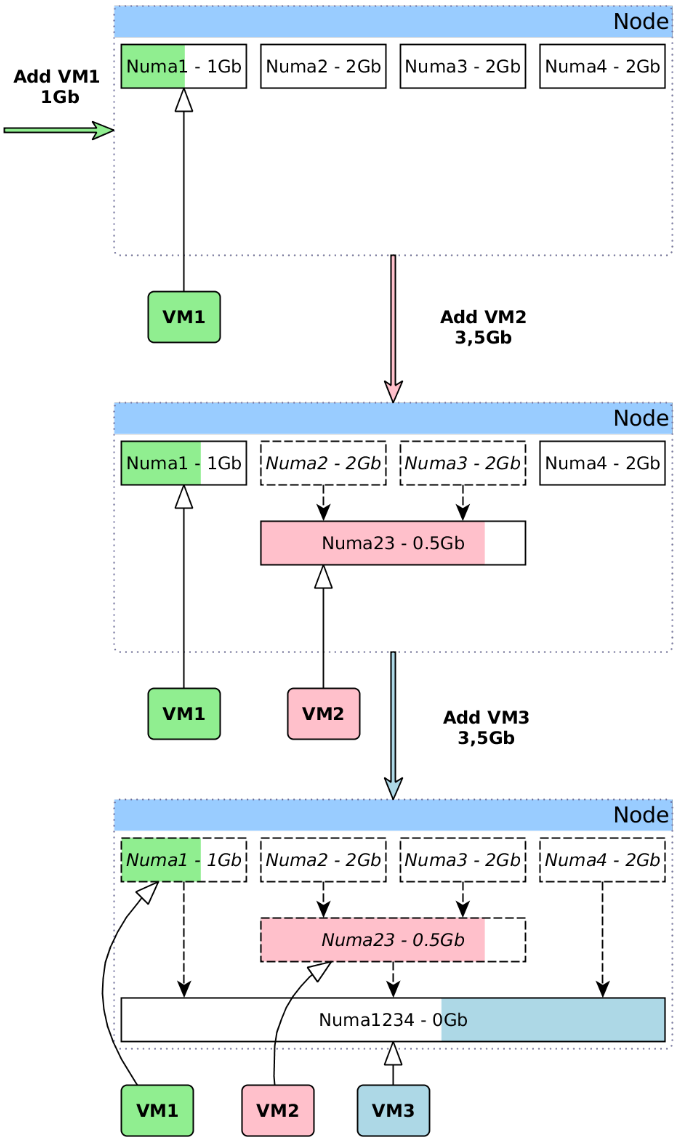


Figure 4.10: NUMA Structure Model with 3 VMs sequentially instantiated.

Figure 4.11 depicts the decomposition phase of our algorithm starting from the previous scenario. First, the termination of vm_2 frees 3.5 Gb memory in $Numa_{2,3}$ and thus in $Numa_{1,2,3,4}$. Despite not having any VM referencing it, $Numa_{2,3}$ is not removed as it still has a parent ($Numa_{1,2,3,4}$). But when vm_3 is terminated, $Numa_{1,2,3,4}$ is neither referenced by any VM nor it has any parent, leading to its removal. The same applies to $Numa_{2,3}$ after removing $Numa_{1,2,3,4}$. At this step, $Numa_1, Numa_2, Numa_3, Numa_4$ can be directly allocated again. The termination of vm_1 returns the physical machine to its initial state.

Allocation Algorithm Now that we have given our composition and decomposition procedure for NUMAs, we detail our VM allocation and deallocation procedure.

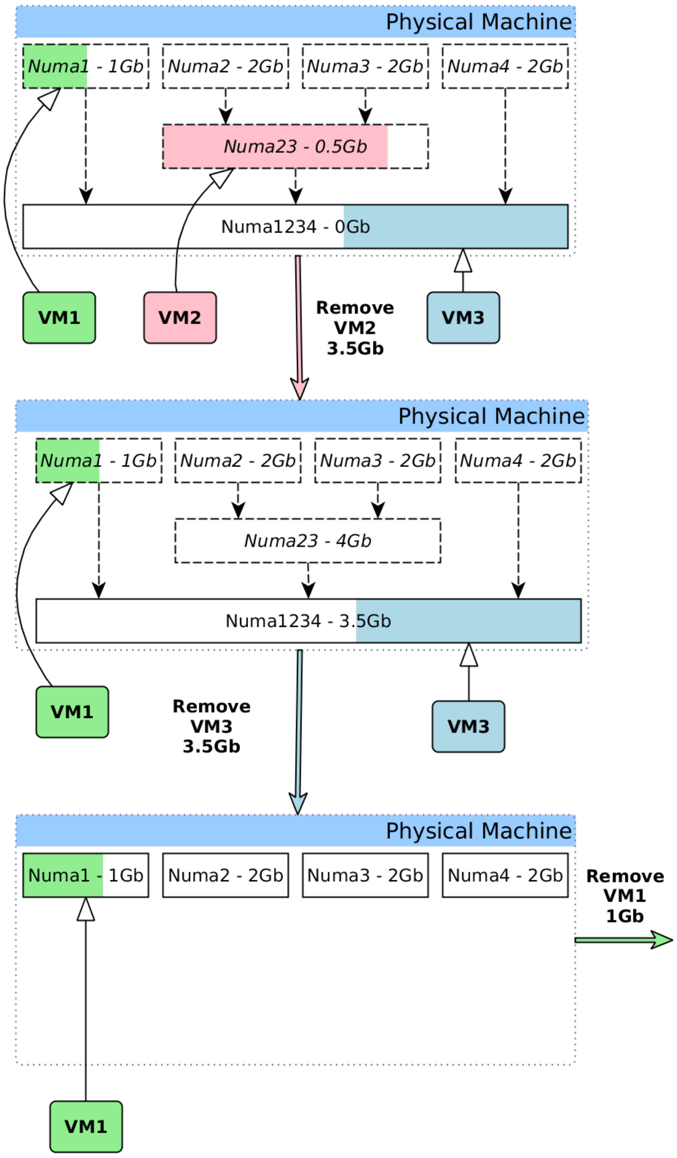


Figure 4.11: NUMA Structure Model with 3 VMs sequentially removed.

Upon the arrival of a new VM, we suppose a set of **Cores** and **VNumas** have been selected according to the information leakage constraints. This selection is simply realized with a First-fit algorithm. In future work, other algorithms such as simulated annealing [71] or genetic algorithms [88] could be used providing a way to compare two configurations (*i.e.*, cores and NUMAs) in order to choose the best one. The complete allocation and deallocation procedures are described in Algorithms 4.6 and 4.7.

Our security-aware allocation is not rendering the placement problem less complex. Indeed, VM placement (and any placement problem) is known to be NP-hard and can be abstracted as bin-packing problem [85]. In the next subsection, we prove that our extension actually increase the complexity of solving the placement problem.

Algorithm 4.6 Allocation Procedure

```
Input: $n$: Node, $v$: VM, $l_{numas}$: List of VNumas , $l_{cores}$:List of Cores
Precondition: $l_{numas} > 0$ {A VM must have some memory}
Variables: $vn$: VNuma
  {Remove cores from availability list}
  $n.cores_{available} \leftarrow n.cores_{available} - l_{cores})$
  {Compose numas if needed and remove sub-numas from availability list}
  if $|l_{numas}| > 1$ then
    $vn \leftarrow$ composeNuma($n$, $l_{numas}$);
  else
    $vn \leftarrow$ getFirst($l_{numas}$) {There is only 1 numa in $l_{numas}$}
  {Substract VM's required memory}
  $vn.mem_{free} - = v.mem_{required}$
  {Increment VM instances}
  $vn.vm_{insts} + = 1$
  {Substract VM's required disk space}
  $n.disk - = v.disk_{required}$
  {Substract VM's required disk space}
  $n.disk - = v.disk_{required}$
  instantiateVM($v$, $l_{cores}$, $vn$)
```

Algorithm 4.7 Deallocation Procedure

```
Input: $n$: Node, $v$: VM, $vn$: VNuma , $l_{cores}$:List of Cores
Variables: $vn$: VNuma
  {Decrement VM instances}
  $vn.vm_{insts} - = 1$
  {Add cores back to availability list}
  $n.cores_{available} \leftarrow n.cores_{available} \cup l_{cores})$
  {Add back VM's required memory}
  $vn.mem_{free} + = v.mem_{required}$
  {Propagate newly free memory to parent NUMAs}
  VNuma $vn_{parent} \leftarrow vn$
  while $(vn_{parent} \leftarrow vn_{parent}.parent) \neq \emptyset)$ do
    $vn_{parent}.mem_{free} + = v.mem_{required}$
  {Decompose if possible}
  decomposeNuma($n$, $vn$);
  {Add vack VM's required disk space}
  $n.disk + = v.disk_{required}$
```

Complexity of our allocation problem

Let an infinity of cores with the VM deployment capacity C. Let a list of virtual machine $\{vm_{1,1}, vm_{1,2}, \ldots, vm_{n,n}\}$. We can split this set of virtual machines, in different sub-set: $\{vm_{1,1}, vm_{1,2}, \ldots, vm_{1,n}\}$ with a size c_n, $\{vm_{2,1}, vm_{2,2}\}$ with a size c_2, and so on. We want pack objects of different volumes (here a sub-set of VMs) into a finite number of bins (here cores) each of volume C in a way that minimizes the number of bins (/cores) used.

We use the binary code to describe the solution and indicate if a VM is on a core or not. Variable x_{ij} is equal to 1 if the VM i is deployed on the core j, and 0 otherwise. Boolean variable y_j is equal to 1 if the core is used, 0 otherwise. We try to minimize the number of core used.

$$min \sum_{j=1}^{n} y_j \tag{4.5}$$

This problem is known as the NP-hard problem called the Bin Packing Problem (BPP). Nevertheless, in our case we have additional constraints due to the security rules. We can consider these constraints like conflicts. The Bin-Packing Problem with Conflicts is called BPPC. It consists in determining the minimal number of identically bins (/cores) needed to store a set of items (VMs) with height is less than the capacity of bins (/cores), where some of these items are incompatible with each other (isolation rules), therefore cannot be packed together in the same bin (/core). The BPPC is a variation of the classical one dimensional BPP which is a combinatorial problem and known also to be NP-hard [37, 84]. Finally, our problem is even harder as we do not have 1 dimension (core) but 2 (memory and core). To keep the proof of concept simple, we rely on a *first-fit* heuristic to solve this BPPC problem.

4.2.4 An Automated Approach

Our approach is a metric-based placement decision. Due to the lack of suitable metric, we have introduced a new information leakage metric measured as a bitrate. Nevertheless, any other sound metric could be substituted. In this subsection, we present our overall automated workflow for VM placement and then discuss how we integrate the concept of metric into our security properties.

Metric-based Placement Decision Workflow

Figure 4.12 depicts our workflow for placement decision. First, we statically compute the risk metric for each host. In the case of our new information leakage metric, we use two tools: Hwloc [24] to obtain the hardware topology and Lmbench[86] to measure the microarchitectural latencies. Both the topology and latencies are needed to compute information leakage values. A tenant provides his virtualized application model comprising isolation properties with risk metrics. Then, we finally decide the configuration (*i.e.,* NUMAs and cores) for each tenant's VM with the placement procedure previously described.

Scenario: Metric computation

Let's exemplify this workflow. We consider two Grid'5000 [5] physical machines called Taurus and Genepi. The topology as obtained from Hwloc were depicted respectively in figures 2.13 and 2.12. We recall that Taurus is a NUMA architecture with a shared L3 cache (15 MB) and Genepi is a SMP architecture with a shared L2 cache (6144 KB).

Cache latencies measured using `lat_mem_rd` from the Lmbench benchmarks suite are presented in Figure 4.13.

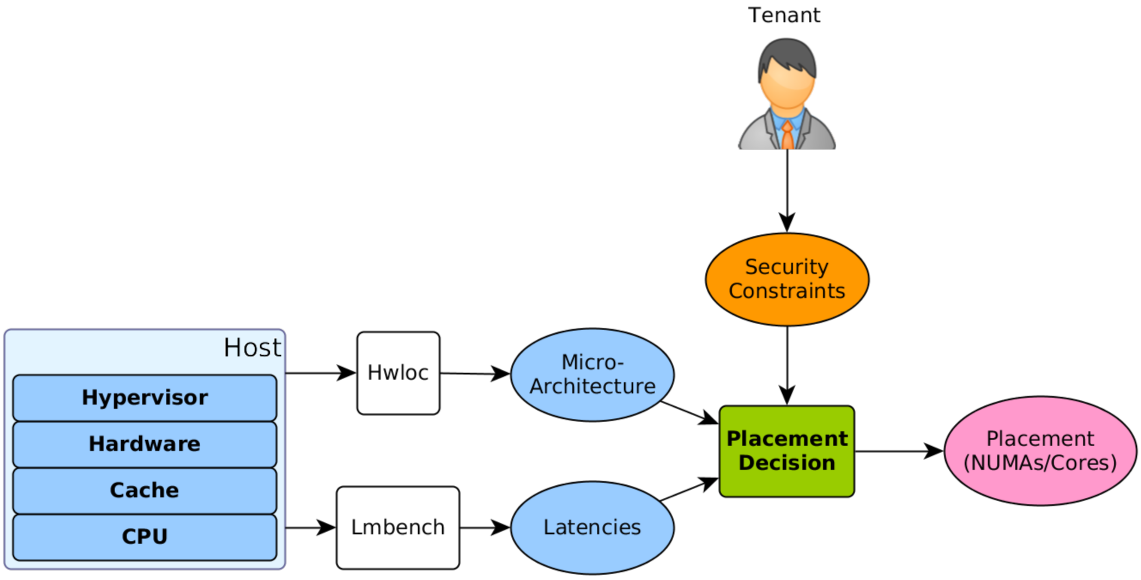


Figure 4.12: Metric-based Placement Decision Workflow.

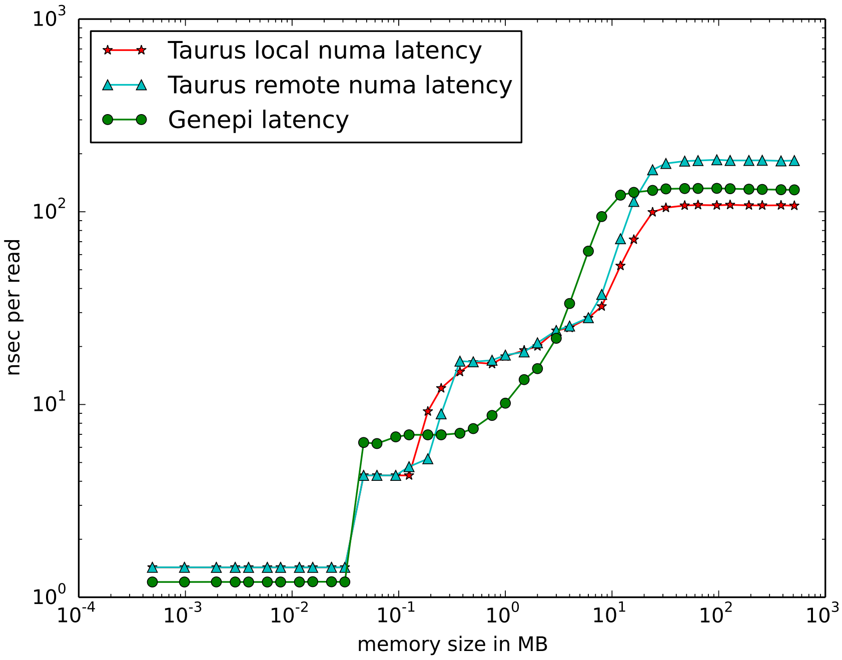


Figure 4.13: Taurus (with NUMAs) and Genepi memory read latencies.

From Taurus latency measures reported in Table 4.2, we can identify 4 steps: 3 cache levels and the main memory (local and remote).

From Genepi latency measures, we can identify 3 steps: 2 cache levels and the main memory.

Finally, using Equation 4.4, Taurus L3 cache-based timing covert channel can transmit one bit in $0.5 \times (17ns + 108ns) \times 15MB/4KB = 240.0\mu s$ *i.e.,* **4.167 Kbps** bitrate on

Component	Block size	latency
L1 cache	1 B to 32 KB	1.43 ns
L2 cache	32 KB to 256 KB	4.3 ns
L3 cache	256 KB to 15 MB	17.0 ns
Local NUMA	-	108 ns
Remote NUMA	-	184.5 ns

Table 4.2: Taurus latency measurements

Component	Block size	latency
L1 cache	1 B to 32 KB	1.2 ns
L2 cache	32 KB to 256 KB	6.3 ns
Main Memory	256 KB to 15 MB	130.0 ns

Table 4.3: Genepi latency measurements

local NUMA, and $0.5 \times (17ns + 184.5ns) \times 15MB/4KB = 386.9\mu s$ *i.e.,* **2.585 Kbps** bitrate on remote NUMA.

Identically, Genepi L2 cache-based timing covert channel can transmit one bit in $0.5 \times (6.3ns + 130ns) \times 6144KB/4KB = 104.7\mu s$ *i.e.,* **9.551 Kbps** bitrate.

Scenario: Tenant specification and Grade

In Section 4.1.3, we have presented our split procedure to obtain typed security properties. In the placement decision, we enforce isolation properties between VMs such as:

```
#property IsolationVM({VM1}, {VM2, VM3}, grade);
```

In Section 2.3.2, we propose a coarse-grain grading system *i.e.,* where *grade* is an integer. If we directly consider the grade to be our information leakage metric, then a bitrate would be inappropriate, if not senseless, for other types of entities like VNet, Data or Service. Indeed, we require the grade to be generic. Therefore, a solution is to consider a direct mapping between grades and information leakage values. The problem of defining the most relevant grades and mapping is part of our future work.

4.3 Automatic Configuration of Security Mechanisms

In addition to having a purely placement-based enforcement of specific security properties (*i.e.,* between VMs), our second complementary solution is the automatic configuration of security mechanisms spread across the infrastructure. This idea is at the core of the Seed4C Project: "seeding" and orchestrating a network of security agents. In this collaborative project, the low-level configuration of security mechanisms comes under our partners and our contribution focuses on the workflow from specification to triggering the low-level configuration.

In the rest of this section, we first present the global architecture of security agents. Then, we detail our solution to integrate security agents into our placement decision.

4.3.1 A Network of Security Agents

All low-level security mechanisms have their own API, language or internal logic. For instance, SELinux policies are composed of *deny* or *allow* rules. Similarly, Iptables can *reject* or *accept* network packets according to condition rules (*e.g.,* accept if the packet comes from a given IP address). The idea is roughly the same in both cases but their configuration have nothing in common. For this reason, a **Security Agent** abstracts these low-level mechanisms. A security agent is a process that automatically translates a security property into a configuration of one or more security mechanisms.

A security agent could be placed virtually anywhere it makes sense. As depicted in Figure 4.14, a security agent may be located in the hypervisor to configure mechanisms at the host level, and/or it may be located at the guest level to manage mechanisms within a virtual machine. The two does not bring the same quality of protection. Whether it is within a VM, we must assume the guest kernel is to be trusted otherwise we cannot guarantee that the agent duly enforces the security requirements. On the other hand, the hypervisor is more trustworthy as the end-user cannot directly influence it. However, protecting entities within VMs is much more limited and complex when using mechanisms at the host level *i.e.,* outside VMs. Security agents may also be placed with network controllers to steer the global network configuration (*e.g.,* allocating VLANs and instantiating L3 overlays). In fact, our previously presented placement-based enforcement could be viewed as a security agent dedicated to enforcing specific properties between VMs.

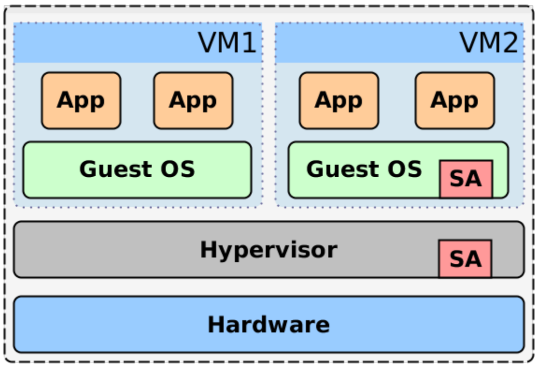


Figure 4.14: Security Agents (SA) Location

4.3.2 Capabilities and Placement Decision

A security agent may enforce a subset of security properties. This subset depends on the available security mechanisms underneath the agent. For example, an agent is able to enforce the confidentiality between Data or Service entities (*e.g.,* with SELinux) but not for other types of entities. This is related to *typed properties* except a *typed property* is first the result of a split procedure and second it is instantiated. Here, the agent has a list of **capabilities** he knows how to enforce. A *capability* is to a security property what a function signature is to a function call. In our example, the capability is:

Confidentiality(Data | Service, Data | Service)

And the following property matches this capability:

```
#property Confidentiality(Data="Log", Service="SSH")
```

A security agent may have multiple mechanisms to enforce the same property and these mechanisms may not provide the same quality of protection. Consequently, capabilities are *graded* and for a capability to match a property, it must have at least the same grade. Therefore, we have the following graded capability:

Confidentiality(Data | Service, Data | Service, 80)

Because the capability offers at least the required quality of protection, it matches the property:

```
#property Confidentiality(Data="Log", Service="SSH", 50)
```

Figure 4.15 depicts our whole placement decision workflow taking into account security agents' capabilities. Security agents can be present at the guest-level *i.e.,* in the VMs of the virtualized application to deploy, and at the host-level *i.e.,* in the hypervisor of some hosts. As illustrated, security agents are not necessarily present in each VM or each host. Each agent has a list of capabilities it can enforce. At the guest-level, this information is attached to the VM image. Indeed, any instance of a VM image have exactly the same mechanisms installed and thus the same list of capabilities. At the host-level, each agent is responsible for publishing its list of capabilities to the placement decision engine. This dynamic approach allows changing hosts' software stack *i.e.,* adding or removing mechanisms (even agents) as needed.

Our decision workflow proceeds as follows. The tenant models its virtualized application and security properties. These properties, after the preprocessing procedure, are sorted into two categories: the capability-based properties with can be enforced by a security agent and the risk-based properties which can only be enforced by selecting correctly hosts and choosing a configuration of NUMAs and cores. Then, a first guest-level solving procedure maps properties to guest-level agents according to their capabilities. After, if some properties are left unsolved, a second host-level solving procedure selects, for each VM, a list of hosts which have the necessary capabilities to solve the VM's properties. Finally, our risk-based placement decision presented in Section 4.2 is applied but with a filtered list of hosts as possiblities. This final step maps each VM to a host and the computed configuration (*i.e.,* NUMAs and cores) as discussed in Section 4.2, but it also maps each VM' properties to the capabilities of their host-level security agents.

4.4 Conclusion

In this chapter, we have first shown the preprocessing steps:

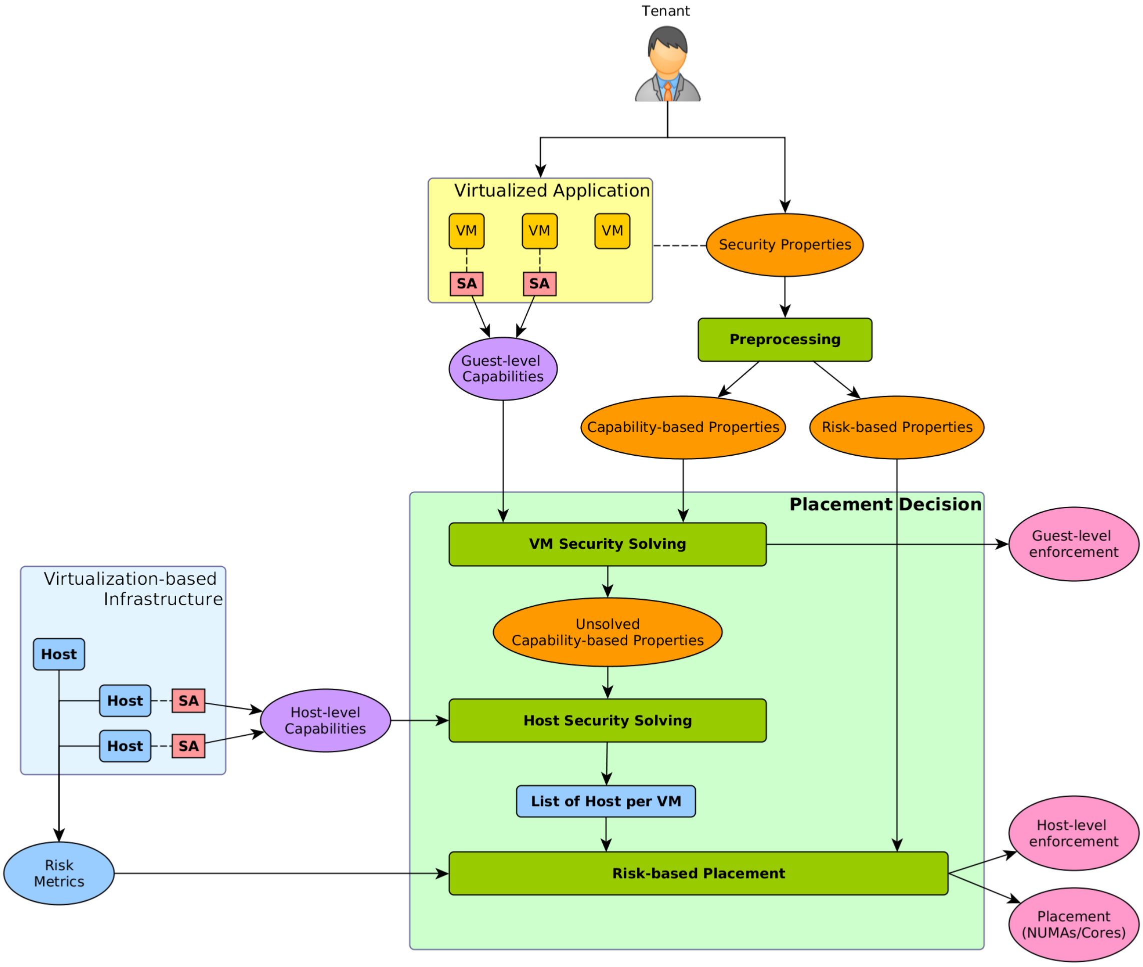


Figure 4.15: Complete Placement Decision Workflow.

1. Implicit to explicit properties: the list of authorized entities can be inferred from the application model to obtain explicit properties.
2. Explicit to singleton properties: a global explicit property can be split into properties with a single secured entity
3. Singleton to typed properties: because mechanisms are specialized, a property can be split into severable properties each one dedicated to a specific type of entity.

Once we obtain typed properties, we have shown two different but complementary methods to enforce them, namely, a placement-based enforcement and the automatic configuration of security mechanisms.

The virtualization cannot provide a strong isolation on shared resources between VMs due to covert channel *i.e.,* leaking unauthorized information by exploiting microarchitectural features. Accordingly, we have first proposed a new information leakage metric to quantify these covert channels, and then an information leakage aware placement algorithm.

The automatic configuration can be achieved with security agents providing capabilities *i.e.,* templates of security properties they can enforce.

In this chapter, we have discussed properties on virtual networks but, due to lack of time, we have not proposed a solution to enforce them. Nevertheless, we could envisioned to tackle this issue as a group membership problem: any VNets (Virtual Networks) with the same properties (and same set of secured/authorized members) can be mapped to the same INet (Infrastructure Network). For instance, VNets without security properties can be mapped to the same *public* INet. Finally, it would be interesting to prove some equivalences for more complex IF-PLTL formulas.

Chapter 5

Use Case: An Advertising Content Manager for Airports

Through this manuscript, we have presented our user-centric approach to propose a security-aware model and deployment of virtualized applications on virtualization-based infrastructure.

In this chapter, we illustrate our framework on an industrial use case: an Advertising Content Manager service for airports. First, we briefly introduce the Ikusi company and its airport management system. Then, we describe the related virtualized application model and security requirements. In Section 5.3, we detail the different steps of our framework: preprocessing, VM security solving, placement-based enforcement and configuration-based enforcement. For the automatic configuration, we briefly describe the actual architecture of security agents as designed by Bousquet *et al.* in [21].

5.1 Ikusi Corporation

Ikusi is a corporation offering a broad range of operational management solutions for smart cities, airports, security, mobility (including railways) and health. In particular, this company proposes airport management systems coupled with public information and entertainment systems. In this chapter, our use case focus on the Advertising Content Manager service for airports called **Musik**.

An airport possesses various spots for advertisements with **devices** like displays or TV screens. Any advertiser can rent one or more spots to broadcast his advertising campaign. This campaign is designed by an **operator** using the *Musik* application. The devices connect to the *Musik* application to get the content they must broadcast.

As depicted Figure 5.1, the *Musik* service is a 3-tier application shared between different airports (*i.e.,* Airport1 and Airport2). Any operators or devices (from any airports) access the application by querying a Web Proxy (*i.e.,* tier 1). For instance, an operator from Airport1 can access his advertising web interface by requesting the address `https://musik.ikusi.com/airport1/design`. Upon receiving this HTTP request, the Web Proxy transfers it to Musik1 App (*i.e.,* tier 2) which returns the web interface. Similarly, a device can query the address `https://musik.ikusi.com/airport2/content` to obtain the advertising content from Musik2 App. If there is one Musik App per airport, all useful data are stored in a common DATABASE (*i.e.,* tier 3) framework called AODB (Airport Operational Database).

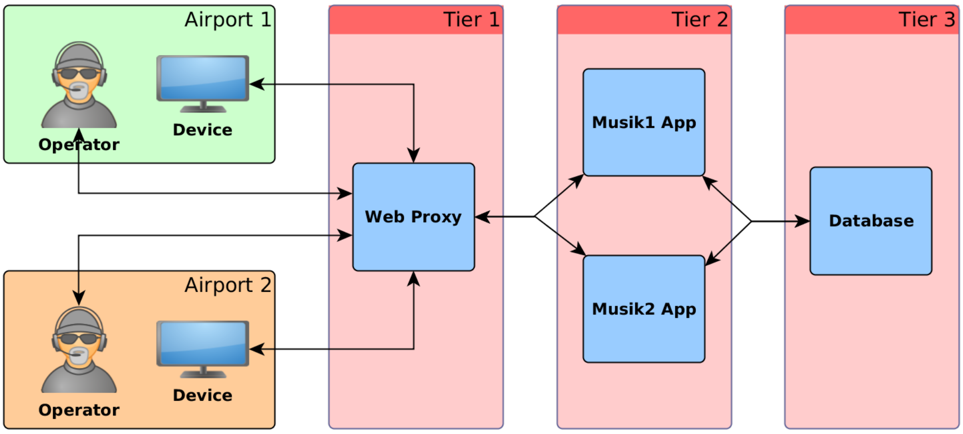


Figure 5.1: Ikusi Advertising Content Manager for Airports

The four components *i.e.,* WEB PROXY, MUSIK1 APP, MUSIK2 APP and DATABASE are executed inside individual virtual machines and deployed on a virtualization-based infrastructure like a Cloud. However, airports' operators and devices are external users of the advertising service.

In the rest of this chapter, we present the Sam4C model of this Advertising Content Manager use case including a meaningful excerpt of the security policy. Then, we detail the successive steps of our approach: preprocessing, deployment and the enforcement of the required security properties using both the placement and the configuration of mechanisms.

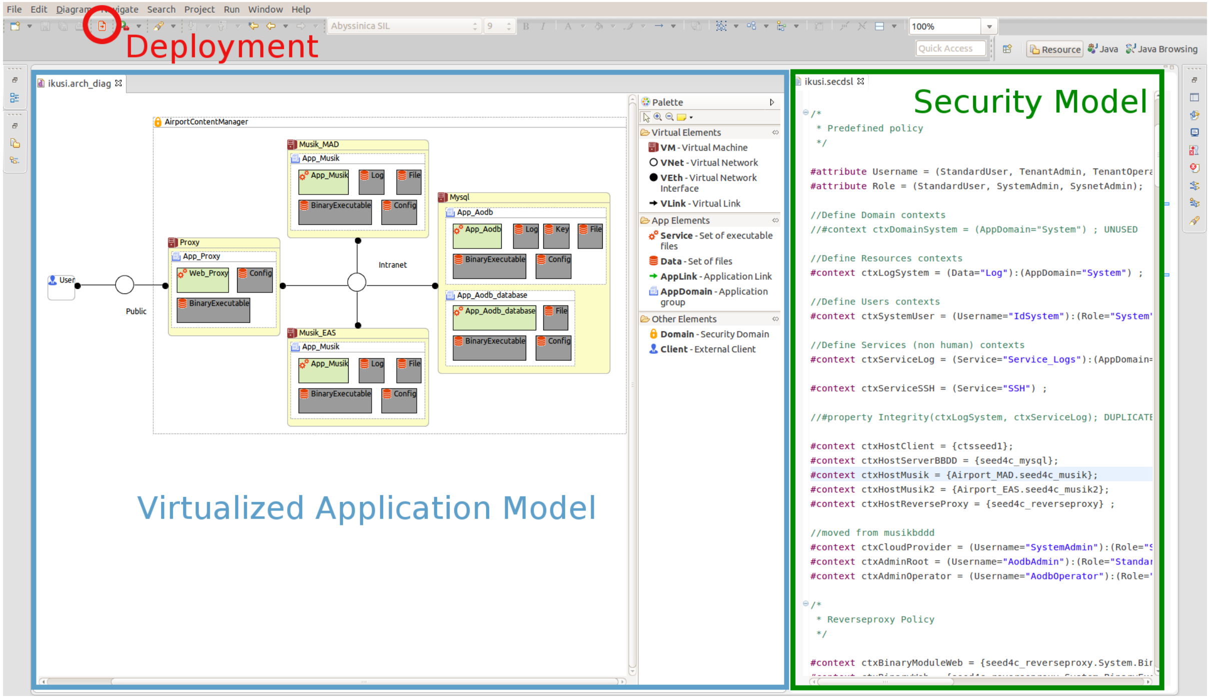
5.2 Modeling

In this section, we present the modelization of the airport use case using the *Sam4C Modeling* tool. A screenshot of the use case model in *Sam4C* is depicted Figure 5.2. The left part contains the virtualized application model with the tool palette to graphically manipule the model. The right part contains the security policy with the definition of attributes, contexts and properties. The circled upper button triggers the deployment of the model by sending it to *Sam4C Scheduler* using its IP address.

In this section, we first detail the virtualized application model and then discuss few security properties.

5.2.1 Virtualized Application Model

The Sam4C Model of the Advertising Content Manager service is depicted Figure 5.3. It is worth noting that the modelization has been done by Ikusi with limited help from our side *i.e.,* Sam4C seems to be easy to use. Two airports are considered namely MAD (for Madrid) and EAS (for San Sebastian). A **Security Domain** AIRPORTCONTENTMANAGER contains the 3-tier application which is composed of 4 VMs:



```
/*
 * Predefined policy
 */

#attribute Username = (StandardUser, TenantAdmin, TenantOpera
#attribute Role = (StandardUser, SystemAdmin, SysnetAdmin);

//Define Domain contexts
//#context ctxDomainSystem = (AppDomain="System") ; UNUSED

//Define Resources contexts
#context ctxLogSystem = (Data="Log"):(AppDomain="System") ;

//Define Users contexts
#context ctxSystemUser = (Username="IdSystem"):(Role="System'

//Define Services (non human) contexts
#context ctxServiceLog = (Service="Service_Logs"):(AppDomain=

#context ctxServiceSSH = (Service="SSH") ;

//#property Integrity(ctxLogSystem, ctxServiceLog); DUPLICATE

#context ctxHostClient = {ctsseed1};
#context ctxHostServerBBDD = {seed4c_mysql};
#context ctxHostMusik = {Airport_MAD.seed4c_musik};
#context ctxHostMusik2 = {Airport_EAS.seed4c_musik2};
#context ctxHostReverseProxy = {seed4c_reverseproxy} ;

//moved from musikbddd
#context ctxCloudProvider = (Username="SystemAdmin"):(Role="S
#context ctxAdminRoot = (Username="AodbAdmin"):(Role="Standa
#context ctxAdminOperator = (Username="AodbOperator"):(Role='

/*
 * Reverseproxy Policy
 */

#context ctxBinaryModuleWeb = {seed4c_reverseproxy.System.Bir
```

Figure 5.2: Screenshot of the Ikusi use case from Sam4C

- Proxy, it is the frontend containing the web proxy. A user (*i.e.*, operator or device) queries the proxy which checks the permissions then forward the HTTP request to the proper Musik Application instance.

- Musik_MAD, the Musik application instance for Madrid airport.

- Musik_EAS, the Musik application instance for San Sebastian airport.

- Mysql, the backend containing the database (App_Aodb_database) and an interface presenting the data to the Musik applications (App_Aodb).

The VMs are interconnected with the Intranet VNet and the web proxy is accessible by a user through the Public VNet.

The four VM's resources (*i.e.*, CPU, RAM, Disk) requirements are given in Table 5.1.

VM	Cores	RAM	Disk
Proxy	1	6 GB	20 GB
Musik_MAD	2	6 GB	50 GB
Musik_EAS	2	6 GB	50 GB
Mysql	1	12 GB	100 GB

Table 5.1: Ikusi VMs Resource Requirements

Each VM contains one or two Application Domains grouping coherent sets of processes and files. For instance, Musik_MAD has an application domain

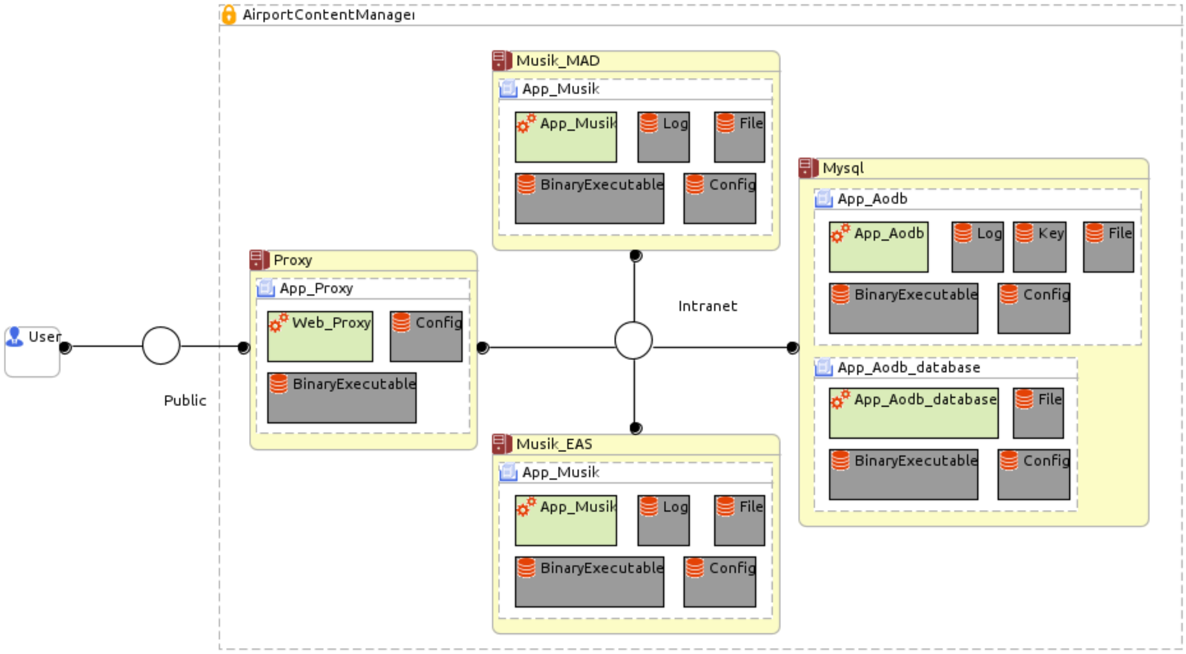


Figure 5.3: Sam4C Model of Ikusi Use Case

App_Musik containing the Service App_Musik itself plus its executable source code (BinaryExecutable), configuration files (Config), the logs generated by the process (Logs) and some files specific to the instance (File).

5.2.2 Security Policy

The complete use case contains more than 70 properties and it would be too exhaustive to review each one. Instead, we selected four of them to illustrate our approach. These security requirements are the following:

1. An external user is authenticated against the web proxy to determine the airport it belongs to and whether it is a device or an operator.

2. Musik MAD application logs can only be modified by the Musik MAD service.

3. Musik MAD application configuration files can only be read by the Musik MAD service.

4. The whole AirportContentManager framework is isolated from any other tenant in the hosting virtualized infrastructure with at least a medium quality.

From these requirements expressed in natural language, we detail afterwards specific attributes and contexts needed to express the security properties plus the mapping between contexts and real resources we refer to as bindings.

Contexts and Bindings

In this use case, an external user can be either a device or an operator and it belongs to either Madrid or San Sebastian airport. Therefore, we define two new attributes: Role and Airport as follows:

```
#attribute Role = (Device, Operator);
#attribute Airport = (MAD, EAS);
```

Listing 5.1: Ikusi Specific Attributes

Proxy Contexts and Bindings For the web proxy VM, we create the alias `ctxProxy` for the application domain App_Proxy. Then, we can simply create a context for the binary source code (`ctxBinaryWeb`) as being the Data element BinaryExecutable in `ctxProxy`.

```
#context ctxProxy = {AirportContentManager.Proxy.App_Proxy};

#context ctxBinaryWeb = ctxProxy:(Data="BinaryExecutable");
#context ctxConfigWeb = ctxProxy:(Data="Config");
```

Listing 5.2: Proxy Contexts Excerpt

These abstract contexts can be associated with “real” component of the system. This association is called *binding*. For instance, `ctxConfigWeb` represents two real files namely `/etc/httpd/conf/httpd.conf` and `/etc/httpd/conf.d/ssl.conf`.

```
"/usr/sbin/httpd" ctxBinaryWeb
"/etc/httpd/conf/httpd.conf" ctxConfigWeb
"/etc/httpd/conf.d/ssl.conf" ctxConfigWeb
```

Listing 5.3: Proxy Bindings Excerpt

Musik Contexts and Bindings The Musik VM is instantiated once per aiport *i.e.*, twice in our case. Nevertheless, as they provide exactly the same service, their security requirements are also the same. As a result, we can optimize the description by defining the contexts and their bindings without specifying which instance they belong to. In the listing below, we give the list of contexts by specifying only the application domain App_Musik.

```
#context ctxBinaryMusik = (Data="Binary|Executable"):(AppDomain="
   App_Musik") ;
#context ctxConfigMusik = (Data="Configuration"):(AppDomain="
   App_Musik") ;
#context ctxFileMusik = (Data="File"):(AppDomain="App_Musik") ;
#context ctxLogMusik = (Data="Log"):(AppDomain="App_Musik") ;
#context ctxServiceMusik = (Service="App_Musik"):(AppDomain="
   App_Musik") ;
```

Listing 5.4: Musik Contexts Excerpt

The following listing contains the corresponding bindings. A binding can be expressed as a regular expression. For instance, "/opt/musik(/.*)?" designates the directory /opt/musik and all files or subdirectories inside.

```
"/opt/musik(/.*)?"      ctxFileMusik
"/opt/musik/properties(/.*)?" ctxConfigMusik
"/opt/musik/musik.lic" ctxConfigMusik
"/opt/musik/rsc(/.*)?" ctxConfigMusik
"/opt/musik/log(/.*)?"     ctxLogMusik
"/opt/musik/webapps/musik.war" ctxBinaryMusik

"/opt/apache-.*/conf/Catalina/localhost(/.*)?" ctxConfigMusik

"/opt/apache-.*(/.*)?" ctxFileMusik
"/opt/apache-.*/bin(/.*)?" ctxBinaryMusik
"/opt/apache-.*/lib(/.*)?" ctxBinaryMusik
"/opt/apache-.*/webapps/*.war" ctxBinaryMusik
"/opt/apache-.*/conf(/.*)?" ctxConfigMusik
"/opt/apache-.*/log(/.*)?" ctxLogMusik

"/opt/devmconn(/.*)?"   ctxFileMusik
"/opt/devmconn/bin(/.*)?" ctxBinaryMusik
"/opt/devmconn/lib(/.*)?" ctxBinaryMusik
"/opt/devmconn/log(/.*)?" ctxLogMusik
"/opt/devmconn/conf(/.*)?" ctxConfigMusik
"/opt/devmconn/properties(/.*)?" ctxConfigMusik

"/opt/ffmpeg(/.*)?"     ctxFileMusik
"/opt/ffmpeg/ffmpeg"    ctxBinaryMusik

"/opt/jre.*(/.*)?"      ctxFileMusik
"/opt/jre.*/bin(/.*)?" ctxBinaryMusik
"/opt/jre.*/lib(/.*)?" ctxBinaryMusik

"/etc/rc\.d/init\.d/activemq"   ctxBinaryMusik
"/etc/rc\.d/init\.d/tomcat"   ctxBinaryMusik
"/etc/rc\.d/init\.d/devmconn"   ctxBinaryMusik

"/opt/jre1.7.0_60/bin/java" ctxServiceMusik
```

Listing 5.5: Musik Bindings Excerpt

Mysql Contexts and Bindings For Mysql VM, we proceed exactly as before by defining two aliases ctxAodbDB and ctxAodb for the two application domains respectively the database and the interface.

```
#context ctxAodbDB = {AirportContentManager.Mysql.
   App_Aodb_database};
```

```
#context ctxBinaryDB = ctxAodbDB:(Data="BinaryExecutable");
#context ctxConfigDB = ctxAodbDB:(Data="Config);
#context ctxFileDB = ctxAodbDB:(Data="File);
#context ctxServiceDB = ctxAodbDB:(Service="App_Aodb_database);

#context ctxAodb = {AirportContentManager.Mysql.App_Aodb};

#context ctxBinayAODB = ctxAodbDB:(Data="BinaryExecutable);
#context ctxConfigAODB = ctxAodbDB:(Data="Config);
#context ctxKeyAODB = ctxAodbDB:(Data="Key);
#context ctxLogAODB = ctxAodbDB:(Data="Log);
#context ctxFileAODB = ctxAodbDB:(Data="File);
#context ctxServiceAODB = ctxAodbDB:(Service="App_Aodb);
```

Listing 5.6: Mysql Contexts Excerpt

Then, the Mysql bindings are the following:

```
"/opt/dbhook(/.*)?"     ctxFileAODB
"/opt/dbhook/dbhook.conf" ctxConfigAODB
"/opt/dbhook/jordi.loc"    ctxConfigAODB
"/opt/dbhook/keys(/.*)?"      ctxKeyAODB
"/opt/dbhook/log(/.*)?"      ctxLogAODB
"/opt/dbhook/lua(/.*)?"      ctxBinayAODB
"/opt/dbhook/pid(/.*)?"      ctxFileAODB
"/opt/dbhook/proxydaemon.sh" ctxBinayAODB
"/opt/dbhook/proxyrun.sh"   ctxBinayAODB
"/opt/dbhook/proxystop.sh"   ctxBinayAODB
"/usr/bin/mysql-proxy"  ctxBinayAODB

"/etc/rc\.d/init\.d/dbhook"   ctxBinayAODB
"/usr/lib64/mysql-proxy/lua/luacall\.so" ctxBinayAODB
"/usr/libexec/mysqld" ctxBinaryDB
"/etc/my\.cnf" ctxConfigDB
"/var/lib/mysql(/.*)?"  ctxFileDB

"/usr/bin/mysqld_safe" ctxServiceDB
"/usr/libexec/mysqld" ctxServiceDB
"/usr/bin/mysql-proxy" ctxServiceAODB
```

Listing 5.7: Mysql Bindings Excerpt

Security Properties

After having described some contexts and their bindings, we express the four security requirements as Sam4C security properties.

First, we model the requirement *“An external user is authenticated against the web proxy to determine the airport it belongs to and whether it is a device or an operator.”* as an Authentication property where any Clients (`Client="*"`) authenticates against the web proxy service in the proxy VM (`ctxProxy:(Service="Web_Proxy")`) to obtain a role attribute (Device or Operator) and an airport attribute (MAD or EAS). From this property, we can express the authorized flows depending on the role and the airport.

```
#property Authentication((Client="*"),
              ctxProxy:(Service="Web_Proxy"),
              [(Role="Device|Operator"):(Airport="MAD|EAS")]);
```

Listing 5.8: Ikusi Authentication Property

Our second and third requirements, namely *"Musik MAD application logs can only be modified by the Musik MAD service."* and *"Musik MAD application configuration files can only be read by the Musik MAD service."*, are respectively an Integrity and a Confidentiality properties. Previously, we have defined Musik contexts without attaching a particular VM. Therefore, in the following listing, we specialize our contexts with the VM Musik_App by writing `(VM="Musik_MAD"):ctx`.

```
#property Integrity((VM="Musik_MAD"):ctxLogMusik,
                    (VM="Musik_MAD"):ctxServiceMusik);
#property Confidentiality((VM="Musik_MAD"):ctxConfigMusik,
                          (VM="Musik_MAD"):ctxServiceMusik);
```

Listing 5.9: Ikusi Integrity and Confidentiality Properties

Finally, the fourth requirement *"The whole AirportContentManager framework is isolated from any other tenant in the hosting virtualized infrastructure with at least a medium quality."* is simply expressed using an implicit property as:

```
#property Isolation({AirportContentManager}, MEDIUM);
```

Listing 5.10: Ikusi Isolation Property

In Section 2.3.2, we introduced our grade as an integer value. But instead of using the full range between 0 and 100, we prefer offering only few levels to the user and have a correspondence between quality levels and grades as detailed in Table 5.2. We use indiscriminately the quality level of grade *i.e.,* `Isolation({AirportContentManager}, MEDIUM)` and `Isolation({AirportContentManager}, 40)` are the same property.

Quality Level	**Grade**
Low	20
Medium	40
High	60
Very High	100

Table 5.2: Correspondence between Quality Levels and Grades

5.3 Deployment

In this section, we detail the deployment phase of our use case. First we preprocess our security properties to obtain locally enforceable properties. Then, we filter the properties that can be enforced by a guest-level security agent. After, we enforce other properties using our placement-based solution. Finally, we present an internal architecture of security agent and show the configurations enforcing the filtered properties.

5.3.1 Preprocessing

In the four properties we have, only one is eligible to be preprocessed namely:

```
#property Isolation({AirportContentManager}, MEDIUM);
```

Listing 5.11: Ikusi Implit Isolation Property

The three other properties secure entities within VMs and thus splitting them is not necessary. For the sake of simplicity, we shorten `AirportContentManager.Entity` into `Entity` and, as grades are preserved through split procedures, each one of the following Isolation properties has implicitly the grade MEDIUM.

First, Algorithm 4.1 (Page 80) extends our *implicit* Isolation property into the two following *explicit* properties:

```
#property Isolation({Musik_MAD, Mysql, Musik_EAS, Intranet},{Proxy
   });
#property Isolation({Proxy},{Public, Musik_MAD, Mysql, Musik_EAS,
   Intranet});
```

Listing 5.12: Ikusi Explicit Isolation Properties

The first explicit property states that PROXY is authorized to exchange information only with MUSIK_MAD, MYSQL, MUSIK_EAS and INTRANET. The second states that any VMs or VNets (in this model) are authorized to exchange information with PROXY.

Then, Algorithm 4.2 (Page 81) transforms our *global* explicit properties into *local* properties *i.e.*, one property per secured entity:

```
#property Isolation({Musik_MAD},{Proxy, Mysql, Musik_EAS, Intranet
   });
#property Isolation({Mysql},{Musik_MAD, Proxy, Musik_EAS, Intranet
   });
#property Isolation({Musik_EAS},{Musik_MAD, Proxy, Mysql, Intranet
   });
#property Isolation({Intranet},{Musik_MAD, Proxy, Mysql, Musik_EAS
   });
#property Isolation({Proxy},{Musik_MAD, Mysql, Musik_EAS, Public,
   Intranet});
```

Listing 5.13: Ikusi Singleton Explicit Isolation Properties

Finally, Algorithm 4.3 (Page 82) split the previous *local* properties into *typed* properties:

```
#property IsolationVNet({Musik_MAD},{Intranet});
#property IsolationVM({Musik_MAD},{Proxy, Mysql, Musik_EAS});
#property IsolationVNet({Mysql},{Intranet});
#property IsolationVM({Mysql},{Proxy, Musik_MAD, Musik_EAS});
```

```
#property IsolationVNet({Musik_EAS},{Intranet});
#property IsolationVM({Musik_EAS},{Proxy, Musik_MAD, Mysql});
#property IsolationVNet({Intranet});
#property IsolationVM({Intranet},{Proxy, Musik_MAD, Mysql,
   Musik_EAS});
#property IsolationVNet({Proxy},{Public, Intranet});
#property IsolationVM({Proxy},{Musik_MAD, Mysql, Musik_EAS});
```

Listing 5.14: Ikusi Typed Isolation Properties

5.3.2 VM Security Solving

Except for our Isolation properties, the three others (*i.e.*, Authentication, Confidentiality and Integrity) may be mapped to a guest-level security agent. In our model, the Proxy and Musik_MAD VMs have the following *capabilities*:

```
Proxy:
  Authentication(Client, Service, Role:Airport)
  Confidentiality(Service, Client)
  Confidentiality(Data, Client)
Musik_MAD:
  Confidentiality(Data, Service)
  Integrity(Data, Service)
```

These capabilities matches our three properties. As a result, the Authentication property is mapped to the Proxy VM security agent and the Confidentiality and Integrity properties are mapped to Musik_MAD VM security agent. Suppose the Authentication capability is not present within the Proxy but any other VM (*e.g.*, Musik_MAD), then because the authentication must be performed by the Web_Proxy service which is in Proxy, no solution would have been found and the deployment would have failed.

5.3.3 Placement-based Enforcement

Our security-aware placement algorithm must enforce the four following properties:

```
#property IsolationVM({Musik_MAD},{Proxy, Mysql, Musik_EAS});
#property IsolationVM({Mysql},{Proxy, Musik_MAD, Musik_EAS});
#property IsolationVM({Musik_EAS},{Proxy, Musik_MAD, Mysql});
#property IsolationVM({Proxy},{Musik_MAD, Mysql, Musik_EAS});
```

Listing 5.15: Ikusi Typed IsolationVM Properties

We recall that all these properties have the Medium grade. We arbitrarily define the correspondence between a grade and a covert channel bitrate in Table 5.3.

Quality Level	Bitrate
Low	<6 Kpbs
Medium	<3 Kbps
High	<1 Kbps
Very High	<100 bps

Table 5.3: Correspondence between Quality Levels and Bitrates

To better illustrate our placement algorithms in particular the NUMA allocation procedure, we consider that our virtualization-based infrastructure is composed of **two Taurus nodes** of Grid'5000. Taurus microarchitecture is depicted in Figure 2.13 (Page 34). We recall that Taurus has 12 cores, 2 NUMAs and a shared L3 cache per NUMA. The achievable cache-based covert channel bitrates for Taurus computed in Section 4.2.4 are presented in Table 5.4.

Memory (NUMA)	Bitrate
Local	4.167 Kbps
Remote	2.585 Kbps

Table 5.4: Taurus L3-based Covert Channel Bitrates

As depicted in Figure 5.4, inially *i.e.,* before deploying the Ikusi use case, *Taurus1* node already executes the VM Other on core $c1$ using 8 GB of memory in $Numa0$ without enforcing any particular security properties.

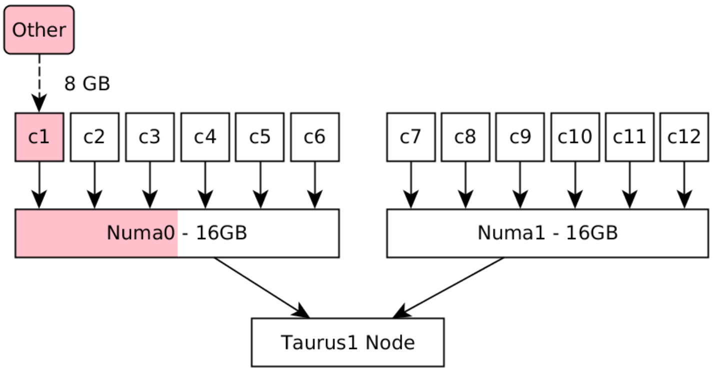


Figure 5.4: Infrastructure Initial State.

To place our virtualized application, we use a First-Fit algorithm: a placement solution is computed for each VM in sequential order (1: Proxy, 2: Musik_MAD, 3: Musik_EAS and 4: Mysql). If a solution exists for all VMs, then the VMs are effectively deployed otherwise the entire application fails.

Because the Medium grade only authorizes less than 3 Kpbs bitrate, the placement configuration cannot have a shared L3 cache with local NUMA in common with an other tenant's VM *e.g.,* Other, but only a remote NUMA. Using our First-Fit placement algorithm in conjunction with Algorithm 4.6 (Page 99), Proxy (1 core, 6 GB memory required) and Musik_MAD (2 cores, 6 GB memory required) are provisioned on the local cores of *Numa*1 *i.e.,* no cache is shared with Other. As a result, we obtain the infrastructure depicted Figure 5.5.

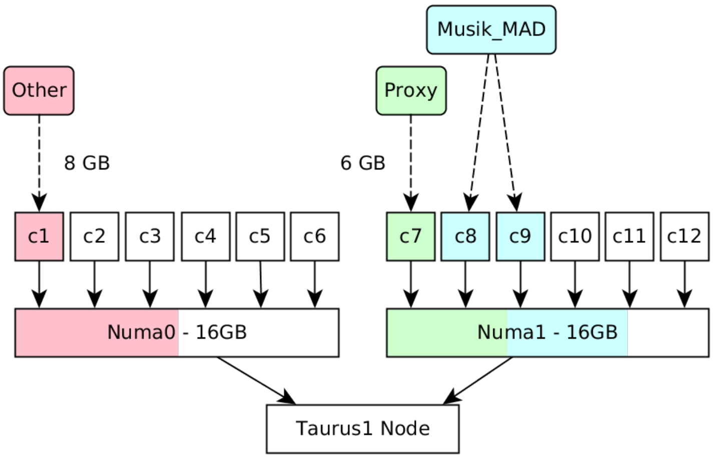


Figure 5.5: Infrastructure State after placing Proxy and Musik_MAD.

At this step, *Numa*0 has 8 GB of available memory and *Numa*1 has 4 GB available memory *i.e.,* a total of 20 GB has been allocated. The next VM in queue *i.e.,* Musik_EAS required 6 GB memory thus it cannot be allocated on *Numa*1 but only on *Numa*0. To enforce Musik_EAS Isolation property, a solution is to allocate the 4 GB available on *Numa*1 and the rest (*i.e.,* 2 GB) on *Numa*0. To satisfy the Medium grade, no local NUMA can be shared with VM Other and thus Musik_EAS is provisioned on cores *c*10 and *c*11. As a result, the NUMA composition procedure detailed in Algorithm 4.4 (Page 96) merges *Numa*0 and *Numa*1 into *Numa*01 and Musik_EAS allocates 6 GB on composed NUMA *Numa*01. The final state of *Taurus1* is depicted in Figure 5.6 where *Numa*01 has 26 GB used including 8 GB by Other on *Numa*0, 6 GB by both Proxy and Musik_MAD on *Numa*1, and 6 GB allocated across *Numa*0 and *Numa*1 by Musik_EAS. At this point, none of the Ikusi VMs share a L3 cache with VM Other hence no cache-based covert channel attack can be conducted (or the bitrate is null)

Finally, Mysql requires 12 GB of memory. With only 8 GB left, *Taurus1* cannot satisfy this constraint. Therefore, our First-Fit algorithm selects *Taurus2* and allocates *c*1 and part of *Numa*0. The final state of *Taurus2* is depicted Figure 5.7.

5.3.4 Configuration-based Enforcement

In Section 5.3.2, we have associated the three properties (*i.e.,* Authentication, Confidentiality and Integrity) to guest-level security agents in VMs Proxy and Musik_MAD. After the deployment of the virtualized application described in Section 5.3.3, some security agents receive a set of properties to enforce. In this subsection, we present the security agent architecture as designed by Bousquet *et al.* in [21]. Then, we detail the enforcement of the Authentication with PAM and the enforcement of the Confidentiality/Integrity with SELinux. Note that any security agent framework could be used providing it implements the correct interface like in [117].

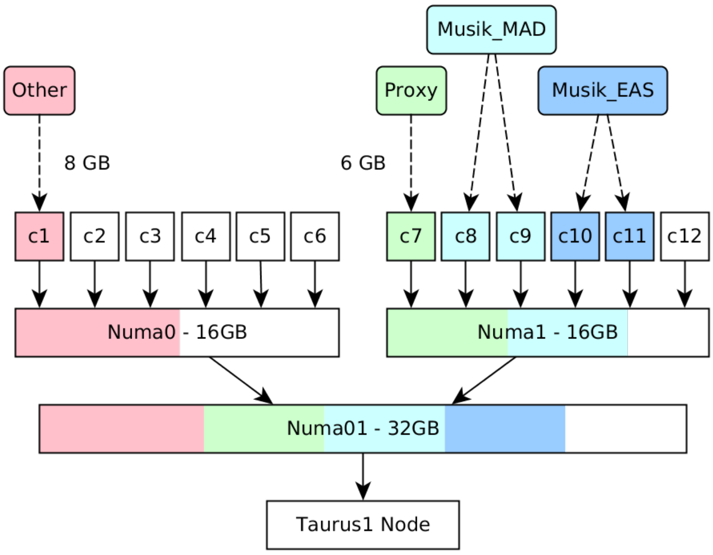


Figure 5.6: Taurus1 Final State.

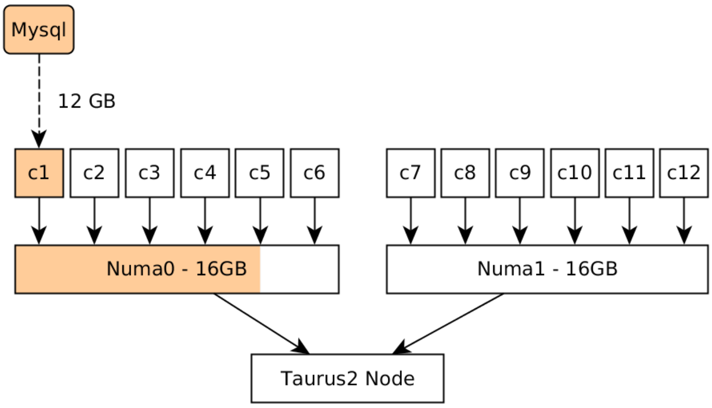


Figure 5.7: Taurus2 Final State.

A Security Agent Design

In [21], a security agent is called a SEE (Secure Element Extended).

As illustrated in Figure 5.8, a security agent has list of *security plugins* which are individual drivers for each security mechanisms. Each plugin publishes the list of capabilities it can enforce to the *capabilities directory*. Upon receiving a list of security properties, a *plugin selector* selects the suitable mechanisms and delegates to the *plugin manager* the orchestration of properly configuring the security mechanisms through their corresponding plugins.

This architecture can encompass any new mechanism by providing the plugin to drive the configuration of the new mechanism and publish the list of its capabilities.

Authentication with PAM

A Pluggable Authentication Module (PAM) is a mechanism to integrate multiple low-level authentication schemes into a high-level application programming interface (API).

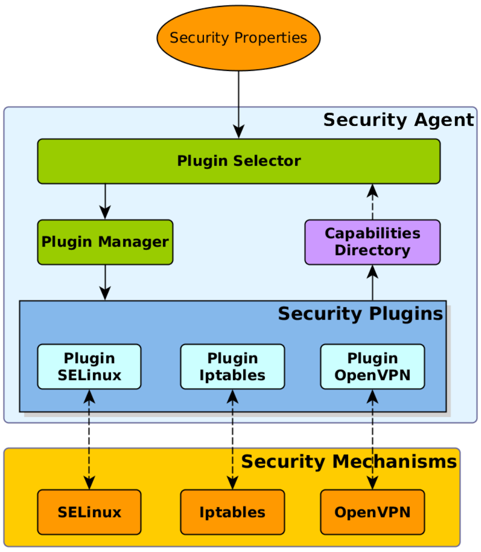


Figure 5.8: Security Agent Architecture.

The security agent in PROXY can enforce the following Authentication property with PAM:

```
#property Authentication((Client="*"),
              ctxProxy:(Service="Web_Proxy"),
              [(Role="Device|Operator"):(Airport="MAD|EAS")]);
```

Listing 5.16: Proxy Authentication Property

Upon receiving this property, the security agent selects the PAM plugin which has the required Authentication capability. This plugin activates an authentication module in PAM like Unix users or LDAP (Lightweight Directory Access Protocol). This module must explicit support the attributes Role and Airport. For instance, Figure 5.9 illustrates an LDAP tree-structure with our two attributes Role and Airport including respectively the groups Device and Operator, and MAD and EAS.

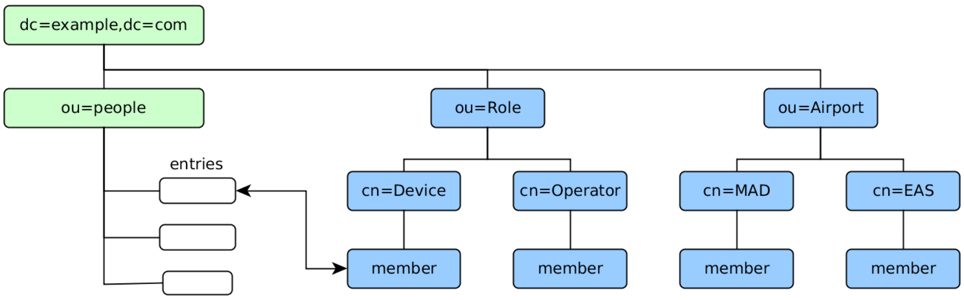


Figure 5.9: LDAP tree structure.

Confidentiality and Integrity with SELinux

The security agent in MUSIK_MAD can enforce the following Confidentiality and Integrity properties with SELinux:

```
#property Integrity((VM="Musik_MAD"):ctxLogMusik,
                    (VM="Musik_MAD"):ctxServiceMusik);
#property Confidentiality((VM="Musik_MAD"):ctxConfigMusik,
                          (VM="Musik_MAD"):ctxServiceMusik);
```

Listing 5.17: Musik_MAD Integrity and Confidentiality Properties

First, the bindings contain all the necessary information to associate *paths* to SELinux contexts. For instance, the generation of a SELinux context for `ctxLogMusik ((Data="Log"):(AppDomain="App_Musik"))` would be:

```
/opt/musik/log(/.*)?      gen_context(system_u:app_musik_r:log_t,s0)
/opt/apache-.*/log(/.*)?  gen_context(system_u:app_musik_r:log_t,s0)
/opt/devmconn/log(/.*)?  gen_context(system_u:app_musik_r:log_t,s0)
```

Then, the Confidentiality property can be translated in SELinux by allowing read operations (everything is denied by default) as follows:

```
interface(‘confidentiality’,‘
gen_require(‘
type $1;
type $2;
’)
allow $1 $2:file { read_file_perms create_file_perms unlink };
allow $1 $2:lnk_file { read_lnk_file_perms create_lnk_file_perms };
allow $1 $2:dir { search create_dir_perms };
’)
```

Finally, the application of SELinux confidentiality is the following function call:

```
confidentiality(app_musik_r:log_t, app_musik_r:app_musik_t)
```

The same process can be realized to enforce the Integrity property by allowing writes operations.

5.3.5 Production Platform Integration

The full deployment implementation is called *Sam4C Scheduler*. It contains all algorithms and procedures previously presented. *Sam4C Scheduler* has been integrated in an open source Cloud software platform software called OpenStack. This lighweight integration consists in:

- Replacing the default OpenStack scheduler by *Sam4C Scheduler* which takes in input *Sam4C* models.
- Extending the OpenStack database with information leakage informations.
- Extending the Openstack database with the available security agents and their capabilities.
- Extending OpenStack agents with new primitives to add/remove security agents/-capabilities.
- Extending OpenStack allocation primitives with the selection of cores and NUMAs.

Given the aforementioned modifications, *Sam4C Scheduler* can infer the infrastructure model from OpenStack database. Upon receiving a request, *Sam4C Scheduler* computes the placement and security solutions as previously described, then sends the local security properties to the security agents and provisioned VMs using the new OpenStack primitives.

5.4 Conclusion

In this chapter, we have demonstrate the usability of *Sam4C* to model a real airport management use case. We have detailed each step of our framework from modelization to enforcement.

Despite this quite simple application architecture, the quantity of information needed to automatically compute this deployment and enforce its security is huge. For example, defining bindings is time-consuming. Moreover, we recall that we only modeled a small part of the complete airport management framework and only detailed 4 out of 70 security properties for this part. Nevertheless, even if not presented in this manuscript, all the 70 properties have been enforced in practice using several mechanisms.

Chapter 6

Conclusions and Perspectives

The evolution of computing platforms could be viewed as a race between demand *i.e.,* what applications/services need to run efficiently, and supply *i.e.,* what processing power infrastructures are able to offer. Initially in the 1960s, huge computers called mainframes were used to run a vast range of services. At the time, security was a primary concern for the military and secondary for the others. Then, with the sophistication of these applications requiring more resources, services were dispatched and interconnected across multiples machines to form clusters. After, with the continuous growing need of computing and storage resources, grid were designed as a federation of resources operated by multiple organizations. Consequently, it has introduced the novel security issue of concurrent users sharing the same infrastructure. Nowadays, any user of a service like Gmail expects to access the application without delay, from anywhere on earth, at any time and in a responsive and efficient manner. Therefore, new infrastructures have been designed to cope with these issues: Clouds. In such virtualization-based distributed environments, services are deployed on infrastructures providing dynamicity, on-demand resources and abstraction of the physical components for portability. Like Grids, these new virtualization-based infrastructures are shared amongst multiple users/tenants from different organizations from which arise new security issues. Instead of facing external threats mainly, internal threats are now considered as much of a risk. But security is often disregarded due to its complexity, costs and absence of tangible benefits. And when we look at current practices, they mostly rely on multiple independent Access Control methods and do not take into account the propagation of the information that is information flows. But with Access Control, once the information is released from its container, a program may (maliciously or unwillingly) leak the information to an unauthorized entity. Moreover, as most providers offer a security by default of the infrastructure, security issues of hosted applications is left to the user. But the task of configuring the security of an application is complex and error-prone. At the opposite, we believe a user should be able to specify its security without any knowledge of available low-level security mechanisms and should benefit from automatic enforcement methods. In resume, we need a user-centric approach as opposed to provider-centric.

As a solution, we have proposed a specification-driven approach where the security is expressed as properties in a mechanism agnostic language. Our specification-driven approach is materialized as a toolbox called *Sam4C* (Security-Aware Models for Clouds).

The first part is *Sam4C Modeling* presented in Chapter 2. We need a simplified representation (*i.e.,* a model) of the application to be able to deploy it and the list of security properties to enforce them. By following a model-driven engineering methodology, the end-user modelization task is rendered easier with a graphical interface for the virtualized application and a textual language for the security requirements. Our modelization focuses on n-tier applications therefore we have defined a virtualized application as a set of interconnected VMs containing data and services. Following our user-centric philosophy, the security requirements are properties regulating the propagation of information between entities modeled in the application. These properties (*e.g.,* Authentication, Confidentiality, Integrity and Isolation) are mechanism-agnostic: they specify what a secure application is but not how to secure it. As the infrastructure provides the means to enforce the security, we proposed an infrastructure metamodel hosting virtualized applications. This metamodel contains both the description of the distributed infrastructure and the microarchitecture of each physical machine composing it.

If the application part of the model is usually well understood, the exact meaning of the security properties is still fuzzy. Besides, the main issue with existing security languages is the ability to formally guarantee the required property. On the one hand, security policies described in a natural language have quite ambiguous semantics. On the other hand, a formal language or logic provides clear syntax and semantics. Moreover, existing mechanisms are dedicated to secure specific type of entities (*e.g.,* VM, Service, Data, VNet). Therefore, the problem is to have a formal definition of security properties and proven procedures to transform the end-user's global security properties into multiple local properties enforceable by several local mechanisms. For these reasons, we proposed a logic language called IF-PLTL (Information Flow Past Linear Time Logic) in Chapter 3. Our logic is dedicated to controlling the propagation of information *i.e.,* direct and indirect information flows. As these information flows cannot be obtained directly, we have explained their construction from low-level observable events. Security decisions are naturally expressed according to past actions. Accordingly, IF-PLTL is based on the past fragment of LTL with the syntax and semantic detailed in Section 3.5. In addition to using IF-PLTL to transform properties, we have proposed a dynamic monitor that can enforce the full expressivity of IF-PLTL even if its complexity (in time and space) would incur a high overhead in practice.

After having a solid formal basis for our security properties, we must consider their enforcement alongside with the deployment of the application they apply to. If multiple approaches may be envisioned to enforce security properties, we propose two different but complementary solutions in Chapter 4. The first one (described in Section 4.2) is a security-aware placement to enforce security between virtual machines (inter-VM security) which may belong to the same or to different applications. The second (described in Section 4.3) is the configuration of security mechanisms to provide security within applications' virtual machines (intra-VM security). But these approaches can only enforce typed properties obtained after the following preprocessing steps detailed in Section 4.1:

1. Implicit to explicit properties: the list of authorized entities can be inferred from the application model to obtain explicit properties.

2. Explicit to singleton properties: a global explicit property can be split into properties with a single secured entity

3. Singleton to typed properties: because mechanisms are specialized, a property can be split into several properties each one dedicated to a specific type of entity.

All these three procedures are based on proven equivalences thanks to IF-PLTL.

In terms of inter-VM security, the virtualization cannot provide a strong isolation on resources shared between VMs due to covert channels which leak unauthorized information by exploiting microarchitectural features. Accordingly, we have proposed a new information leakage metric to quantify these covert channels, and then integrated this metric into our placement algorithm. The idea is to let the user specify the risk it is willing to face and strictly enforce this constraint by carefully allocating cores and NUMAs for each VM. For intra-VM security, the automatic configuration of mechanisms has been achieved with security agents providing capabilities *i.e.,* templates of security properties they can enforce. Consequently, the problem has been reduced to choosing security agents with the right capabilities.

To illustrate our whole framework, we have presented a real industrial use case: An Advertising Content Manager for Airports in Chapter 5. Using *Sam4C Modeling*, Ikusi's engineers have modeled their application and security requirements with limited external help *i.e., Sam4C* seems to be easy-to-use. Each method detailed in each chapter has been exemplified on this use case. In our opinion, this use case demonstrates two general points. First, the task of providing an end-to-end automatic security a real application is not trivial. Many layers from hardware to software code are involved and we are far from providing a precise and complete modelization of the application. But our second point is that it is feasible to do so by relying on formal methods and integrating good practices from domains other than security like model-driven engineering.

6.1 Short-Term Perspectives

Our contributions can be extended in several directions. In this section, we present our short-term perspectives.

Modelization The current *Sam4C* implementation can be improved in several ways.

- First, we can provide automatic analysis tools to verify the validity of the modelization *i.e.,* that all the needed information are specified for a deployment or that the security properties are not contradictory.

- Second, our current metamodel focuses mainly on the IaaS layer with VMs and VNets; it would be interesting to extend our metamodel to the PaaS and SaaS layers by providing the corresponding models and propose a cross-layer security from the hypervisor to application-specific flows.

- Third, we can study new use cases to validate our metamodel or extend it with the necessary features.

- And last but not the least, our current deployment is limited to provisioning VMs with components statically installed in those VMs *i.e.,* we suppose that each VM contains the correct software stack. The full deployment of all components composing this software stack could be automatized as well. To do so, we believe that component-based modeling is a key lead bringing more adaptability, dynamicity and flexibility to our models. An automatic deployment of inner components can facilitate the enforcement of the security. Indeed, without automaticity, the enforcement must be based on the existing architecture which must be accurately specified by the end-user whereas in an automatic process, all these information can be known without requiring the end-user's knowledge.

Logic In our formalization, we have presented a dynamic monitor. It would be interesting to implement it at the system level and evaluate its overhead in practice with a realistic security policy.

Security Enforcement Regarding the enforcement of security, several points can be addressed:

- First, our placement decision is constrained by covert channels. These constraints lead to a deconsolidation of the platform *i.e.,* VMs are more likely to be spread across the infrastructure than be regrouped on the same host. This side effect is in opposition with the economical benefit of maximizing the consolidation *i.e.,* security costs. Therefore, a cost-model for covert channels should be devised to quantify and monetize this impact.

- Then, we have presented the enforcement of properties between VMs and within VMs but we have left aside the enforcement of network properties. We have discussed a potential solution in Section 4.4 and more effort should be put to explore this direction as securing networks is as important as securing VMs.

- Finally, we have briefly discussed the necessity of having a grading system in Section 2.3.2. A short-term contribution would be to effectively propose a user-friendly grading system and assess its suitability by conducting a survey on a panel of users.

6.2 Long-Term Perspectives

After having presented our short-term perspectives, we discuss in this section our long-term perspectives.

Modelization We have previously proposed to integrate component-based models in our modelization to automatically deploy the software stack within VMs. Following this idea, component-based models are well known to represent dynamic applications scaling at runtime, for instance by changing the number of instances of a VM delivering a service depending on the load. This approach can be further extended to encompass applications which requires some reconfiguration at runtime, for instance to update a legacy architecture with a more efficient but completely different one. This direction should be explored alongside a dynamically reconfigurable security detailed later.

Logic Our logic formalism can be improve in several ways:

- Thanks to IF-PLTL, we have proved few equivalences specifically for the Confidentiality, Integrity and Isolation properties, but we lack general methods to obtain local properties *i.e.,* methods taking as input any arbitrary IF-PLTL formula.
- Furthermore, to have a complete approach, we should provide the means to ensure the correctness of a security policy *i.e.,* prove that a given theory (policy) is coherent. Indeed, allowing an end-user to deploy an incoherent policy is undesirable.
- Our current monitor is not deadlock-free *i.e.,* a sequence of events can lead to forbidding any future accesses. Consequently, the secured system is overprotected as it does not render any service anymore. The desirable property for a monitor is to preserve the liveness while making decisions. One idea is to detect any events leading to deadlock situation and prevent them.

Security Enforcement As stated before, security has a cost but as demonstrated in this Thesis, a general security is hard to quantify or qualify. We can envision two directions:

- A generic and precise metric or grading system for security. The first milestone is to propose a metric to compare two security configurations *i.e.,* being able to have a partial order on security configurations. Then, the second step is having a metric scale proportional/coherent with the security risk.
- The previous proposition can facilitate the design of a general cost-model associating a price to a given security configuration.

Distributed Security If many works have been conducted on system security, programming language security and network security, fewer achievements have been obtained on what we call distributed security. Current solutions revolve around a more or less centralized security of a distributed system and not a fully distributed security. The closest concept would be decentralized security with is divided into Decentralized Access Control [74] and Decentralized Information Flow Control. In AC, the decentralized component is the delegation authority *i.e.,* the entity delivering roles and permissions, and controls are locally authorized, for instance a distributed/decentralized Public Key Infrastructure [81]. In IFC, the major technique is tainting which piggybacks the security policy on each entity. But this technique is efficient only with lightweight piggybacked information and has mostly been applied to a single machine. If we consider a large distributed system with much more entities and conflicting domains, this information grows too and may not scale. Besides, tainting techniques have only been applied only for Confidentiality and Integrity properties. Therefore, we lack scalable methods to control complex information flow properties across a distributed system like Clouds. Generally speaking, the security of each individual components does not ensure the security of the whole. Accordingly, a first can be the design of composable security properties to prove the equivalence between locally enforced properties and a global property of the system. A collaborative security system could also be interesting to explore.

Autonomic Security A very interesting perspective is autonomic security. It can be characterized as a security that can reconfigure itself, recover from intrusions, dynamically adapt the security to the current state of the system. Currently, the closest concept is autonomic networking which includes some considerations on security, for instance by learning from a DDoS network attack to update the firewall rules accordingly. However, the autonomic security has yet to be designed for systems (*e.g.,* OSes).

Availability and Fault-tolerance In this Thesis, we have deliberately ignored the Availability property of the CIA triad as it comes under the concept of fault-tolerance. A denial of service is a common attack which violates the Availability but neither the Confidentiality nor the Integrity. Yet, it is a threat any security administrator must cope with. In our opinion, security and fault-tolerance should be more integrated. Indeed, a byzantine fault (in the fault-tolerance domain) can be seen as a malicious user. Therefore, the study of byzantines may help designed more robust security systems.

Appendix A

Annex

A.1 Publications

A.1.1 Journal

- S. Betgé-Brezetz, A. Bousquet, J. Briffaut, E. Caron, L. Clevy, M.-P. Dupont, G.-B. Kamga, J.-M. Lambert, A. Lefray, B. Marquet, J. Rouzaud-Cornabas, L. Toch, C. Toinard, and B. Venelle. Seeding the cloud: An innovative approach to grow trust in cloud based infrastructures. In A. Galis and A. Gavras, editors, *The Future Internet*, volume 7858 of *Lecture Notes in Computer Science*, pages 153–158. Springer Berlin Heidelberg, 2013

A.1.2 Book Chapter

- M. Blanc, A. Bousquet, J. Briffaut, L. Clevy, D. Gros, A. Lefray, J. Rouzaud-Cornabas, C. Toinard, and B. Venelle. Mandatory access protection within cloud systems. In S. Nepal and M. Pathan, editors, *Security, Privacy and Trust in Cloud Systems*, pages 145–173. Springer Berlin Heidelberg, 2014

A.1.3 International Conferences

- E. Caron, A. D. Le, A. Lefray, and C. Toinard. Definition of security metrics for the cloud computing and security-aware virtual machine placement algorithms. In *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2013*, pages 125–131, Oct. 2013

- L. Bobelin, A. Bousquet, J. Briffaut, J.-F. Couturier, C. Toinard, E. Caron, A. Lefray, and J. Rouzaud-Cornabas. An advanced security-aware cloud architecture. In *High Performance Computing & Simulation (HPCS), 2014 International Conference on*, pages 572–579. IEEE, 2014

- A. Lefray, E. Caron, J. Rouzaud-Cornabas, and C. Toinard. Microarchitecture-aware virtual machine placement under information leakage constraints. In *IEEE 8th International Conference on Cloud Computing (CLOUD), 2015*, pages 588–595, June 2015

A.1.4 National Conference

- A. Lefray and J. Rouzaud-Cornabas. Formalisation de propriétés de flux d'information avec une logique temporelle du premier ordre pour assurer la sécurité d'une infrastructure de cloud. In *Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS), 2014*, 2014

A.1.5 Poster and Talk

- E. Caron, A. Lefray, B. Marquet, and J. Rouzaud-Cornabas. A cloud security infrastructure validated on Grid'5000. Talk at Grid'5000 Winter School., 2012

- A. Lefray, E. Caron, J. Rouzaud-Cornabas, H. Zhang, A. Bousquet, J. Briffaut, and C. Toinard. Security-aware models for clouds. Poster at the 22th IEEE International Symposium on High Performance Distributed Computing (HPDC), Jun. 2013

Bibliography

[1] B. Alpern and F. B. Schneider. Defining liveness. *Information processing letters*, 21(4):181–185, 1985.

[2] B. Atkinson, G. Della-Libera, S. Hada, M. Hondo, P. Hallam-Baker, J. Klein, B. LaMacchia, P. Leach, J. Manferdelli, H. Maruyama, et al. Web services security (ws-security). *Specification, Microsoft Corporation*, 2002.

[3] C. Atkinson and T. Kuhne. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003.

[4] J. Bacon, K. Moody, and W. Yao. A model of OASIS role-based access control and its support for active security. *ACM Trans. Inf. Syst. Secur.*, 5(4):492–540, Nov. 2002.

[5] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.

[6] S. Barker and P. J. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Trans. Inf. Syst. Secur.*, 6(4):501–546, Nov. 2003.

[7] L. Barroso, J. Dean, and U. Holzle. Web search for a planet: The Google cluster architecture. *Micro, IEEE*, 23(2):22–28, March 2003.

[8] D. Basin, M. Clavel, and M. Egea. A decade of model-driven security. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies*, SACMAT '11, pages 1–10, New York, NY, USA, 2011. ACM.

[9] D. Basin, F. Klaedtke, and S. Müller. Policy monitoring in first-order temporal logic. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, number 6174 in Lecture Notes in Computer Science, pages 1–18. Springer Berlin Heidelberg, Jan. 2010.

[10] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, Sept. 2011.

[11] M. Becker and P. Sewell. Cassandra: distributed access control policies with tunable expressiveness. In *Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, POLICY 2004*, pages 159–168, 2004.

[12] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical report, The MITRE Corporation, 1973.

[13] K. Bernsmed, M. Jaatun, P. Meland, and A. Undheim. Security SLAs for federated cloud services. In *Sixth International Conference on Availability, Reliability and Security (ARES), 2011*, pages 202–209, Aug 2011.

[14] C. Bertolissi and W. Uttha. Automated analysis of rule-based access control policies. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*, PLPV '13, page 47–56, New York, NY, USA, 2013. ACM.

[15] S. Betgé-Brezetz, A. Bousquet, J. Briffaut, E. Caron, L. Clevy, M.-P. Dupont, G.-B. Kamga, J.-M. Lambert, A. Lefray, B. Marquet, J. Rouzaud-Cornabas, L. Toch, C. Toinard, and B. Venelle. Seeding the cloud: An innovative approach to grow trust in cloud based infrastructures. In A. Galis and A. Gavras, editors, *The Future Internet*, volume 7858 of *Lecture Notes in Computer Science*, pages 153–158. Springer Berlin Heidelberg, 2013.

[16] K. J. Biba. Integrity considerations for secure computer systems. Technical report, The MITRE Corporation, 1977.

[17] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner. OpenTOSCA – a runtime for TOSCA-based cloud applications. In S. Basu, C. Pautasso, L. Zhang, and X. Fu, editors, *Service-Oriented Computing*, volume 8274 of *Lecture Notes in Computer Science*, pages 692–695. Springer Berlin Heidelberg, 2013.

[18] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann. TOSCA: Portable automated deployment and management of cloud applications. In A. Bouguettaya, Q. Z. Sheng, and F. Daniel, editors, *Advanced Web Services*, pages 527–549. Springer New York, 2014.

[19] M. Blanc, A. Bousquet, J. Briffaut, L. Clevy, D. Gros, A. Lefray, J. Rouzaud-Cornabas, C. Toinard, and B. Venelle. Mandatory access protection within cloud systems. In S. Nepal and M. Pathan, editors, *Security, Privacy and Trust in Cloud Systems*, pages 145–173. Springer Berlin Heidelberg, 2014.

[20] L. Bobelin, A. Bousquet, J. Briffaut, J.-F. Couturier, C. Toinard, E. Caron, A. Lefray, and J. Rouzaud-Cornabas. An advanced security-aware cloud architecture. In *High Performance Computing & Simulation (HPCS), 2014 International Conference on*, pages 572–579. IEEE, 2014.

[21] A. Bousquet, J. Briffaut, and C. Toinard. An autonomous cloud management system for in-depth security. In *IEEE 3rd International Conference on Cloud Networking (CloudNet), 2014*, pages 368–374, Oct 2014.

[22] L. Bozzelli, M. Křetínský, V. Řehák, and J. Strejček. On decidability of LTL model checking for process rewrite systems. *Acta Informatica*, 46(1):1–28, Feb. 2009.

[23] J. Briffaut, J.-F. Lalande, and C. Toinard. Formalization of security properties: enforcement for MAC operating systems and verification of dynamic MAC policies. *International journal on advances in security*, 2(4):325–343, 2009.

[24] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In *18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2010*, pages 180–186, Feb 2010.

[25] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[26] G. Bruns and M. Huth. Access control via belnap logic: Intuitive, expressive, and analyzable policy composition. *ACM Trans. Inf. Syst. Secur.*, 14(1):9:1–9:27, June 2011.

[27] R. Butler, V. Welch, D. Engert, I. Foster, S. Tuecke, J. Volmer, and C. Kesselman. A national-scale authentication infrastructure. *Computer*, 33(12):60–66, Dec 2000.

[28] E. Caron, F. Desprez, and J. Rouzaud-Cornabas. Smart resource allocation to improve cloud security. In S. Nepal and M. Pathan, editors, *Security, Privacy and Trust in Cloud Systems*, pages 103–143. Springer Berlin Heidelberg, 2014.

[29] E. Caron, A. D. Le, A. Lefray, and C. Toinard. Definition of security metrics for the cloud computing and security-aware virtual machine placement algorithms. In *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2013*, pages 125–131, Oct. 2013.

[30] E. Caron, A. Lefray, B. Marquet, and J. Rouzaud-Cornabas. A cloud security infrastructure validated on Grid'5000. Talk at Grid'5000 Winter School., 2012.

[31] E. Caron and J. Rouzaud-Cornabas. Improving users' isolation in IaaS: Virtual machine placement with security constraints. In *7th IEEE International Conference on Cloud Computing, IEEE CLOUD 2014*, Anchorage, USA, June 27-July 2 2014. IEEE Computer Society.

[32] A. Celesti, F. Tusa, M. Villari, and A. Puliafito. How to enhance cloud architectures to enable cross-federation. In *IEEE 3rd International Conference on Cloud Computing (CLOUD), 2010*, pages 337–345, July 2010.

[33] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking.* MIT press, 1999.

[34] M. Clarkson, B. Finkbeiner, M. Koleini, K. Micinski, M. Rabe, and C. Sánchez. Temporal logics for hyperproperties. In M. Abadi and S. Kremer, editors, *Principles of Security and Trust*, volume 8414 of *Lecture Notes in Computer Science*, pages 265–284. Springer Berlin Heidelberg, 2014.

[35] M. R. Clarkson and F. B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, Sept. 2010.

[36] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software testing and analysis*, pages 196–206. ACM, 2007.

[37] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for NP-hard problems. chapter Approximation Algorithms for Bin Packing: A Survey, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997.

[38] M. Da Silva, D. Ardagna, N. Ferry, and J. Perez. Model-driven design of cloud applications with quality-of-service guarantees: The modaclouds approach, micas tutorial. In *16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2014*, pages 3–10, Sept 2014.

[39] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In M. Sloman, E. Lupu, and J. Lobo, editors, *Policies for Distributed Systems and Networks*, volume 1995 of *Lecture Notes in Computer Science*, pages 18–38. Springer Berlin Heidelberg, 2001.

[40] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan. Side-channel vulnerability factor: A metric for measuring information leakage. In *39th Annual International Symposium on Computer Architecture (ISCA), 2012*, pages 106–117, June 2012.

[41] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[42] J. DeTreville. Binder, a logic-based security language. In *Proceedings of the IEEE Symposium on Security and Privacy, 2002*, pages 105 – 113, 2002.

[43] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 17–30, New York, NY, USA, 2005. ACM.

[44] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[45] X. Etchevers, T. Coupaye, F. Boyer, and N. de Palma. Self-configuration of distributed applications in the cloud. In *IEEE International Conference on Cloud Computing (CLOUD), 2011*, pages 668–675, July 2011.

[46] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. 1974.

[47] J. Ferreirós. The road to modern logic—an interpretation. *Bulletin of Symbolic Logic*, 7:441–484, 12 2001.

[48] M. Fonda, C. Toinard, and S. Moinard. Mandatory access control for web applications and workflows. In *The 2013 International Conference on Security and Management (SAM'13)*, 2013.

[49] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational Grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, CCS '98, pages 83–92, New York, NY, USA, 1998. ACM.

[50] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, Aug. 2001.

[51] J. A. Goguen and J. Meseguer. *Security Policies and Security Models*, volume 0. IEEE Computer Society, Los Alamitos, CA, USA, 1982.

[52] A. Gregory. Foundations of multithreaded, parallel, and distributed programming. *ISBN*, 201357526:27–69, 2000.

[53] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[54] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying information flow goals in security-enhanced linux. *Journal of Computer Security*, 13(1):115–134, 2005.

[55] J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying information flow goals in security-enhanced linux. *J. Comput. Secur.*, 13(1):115–134, 2005.

[56] J. Haigh and W. Young. Extending the noninterference version of MLS for SAT. *IEEE Transactions on Software Engineering*, 13(2):141–150, 1987.

[57] J. Y. Halpern and V. Weissman. Using first-order logic to reason about policies. *ACM Trans. Inf. Syst. Secur.*, 11(4):21:1–21:41, July 2008.

[58] S. Hansson. Ideal worlds — wishful thinking in deontic logic. *Studia Logica*, 82(3):329–336, 2006.

[59] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, Aug. 1976.

[60] R. R. Henning. Security service level agreements: Quantifiable security for the enterprise? In *Proceedings of the 1999 Workshop on New Security Paradigms*, NSPW '99, pages 54–60, New York, NY, USA, 2000. ACM.

[61] B. Hicks, S. Rueda, L. St.Clair, T. Jaeger, and P. McDaniel. A logical specification and analysis for SELinux MLS policy. *ACM Trans. Inf. Syst. Secur.*, 13(3):26:1–26:31, July 2010.

[62] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of computer security*, 6(3):151–180, 1998.

[63] S. S. Huang, T. J. Green, and B. T. Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1213—-1216, 2011.

[64] D. Jayasinghe, C. Pu, T. Eilam, M. Steinder, I. Whally, and E. Snible. Improving performance and availability of services hosted on IaaS clouds with structural constraint-aware virtual machine placement. In *IEEE International Conference on Services Computing (SCC), 2011*, pages 72–79, July 2011.

[65] D. Jayasinghe, C. Pu, T. Eilam, M. Steinder, I. Whally, and E. Snible. Improving Performance and Availability of Services Hosted on IaaS Clouds with Structural Constraint-aware Virtual Machine Placement. In *IEEE International Conference on Services Computing (SCC), 2011*, pages 72–79. IEEE, 2011.

[66] H. Jin, W. Qiang, X. Shi, and D. Zou. VO-sec: An access control framework for dynamic virtual organization. In *Proceedings of the 10th Australasian Conference on Information Security and Privacy*, ACISP'05, pages 370–381, Berlin, Heidelberg, 2005. Springer-Verlag.

[67] D. Kafura and D. Gracanin. An information flow control meta-model. In *Proceedings of the 18th ACM Symposium on Access Control Models and Technologies*, SACMAT '13, page 101–112, New York, NY, USA, 2013. ACM.

[68] K. Kahley, M. Radhakrishnan, and J. Solworth. Factoring high level information flow specifications into low level access controls. In *Fourth IEEE International Workshop on Information Assurance, IWIA 2006*, pages 17 pp.–186, April 2006.

[69] A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1):57–81, 2003.

[70] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. NoHype: Virtualized Cloud Infrastructure without the Virtualization. *SIGARCH Comput. Archit. News*, 38(3):350–361, June 2010.

[71] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[72] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann. Winery – a modeling tool for TOSCA-based cloud applications. In S. Basu, C. Pautasso, L. Zhang, and X. Fu, editors, *Service-Oriented Computing*, volume 8274 of *Lecture Notes in Computer Science*, pages 700–704. Springer Berlin Heidelberg, 2013.

[73] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 321–334, New York, NY, USA, 2007. ACM.

[74] J. Laganier and P.-B. Primet. Hipernet: a decentralized security infrastructure for large scale grid environments. In *The 6th IEEE/ACM International Workshop on Grid Computing, 2005*, pages 8 pp.–, Nov 2005.

[75] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190—-222, Apr. 1983.

[76] B. Lampson. Protection. In *Proc. 5th Princeton Conf. on Information Sciences and Systems*, pages 18–24. Princeton, 1971.

[77] J.-C. Laprie. Dependable computing and fault-tolerance. *Digest of Papers FTCS-15*, pages 2–11, 1985.

[78] A. Lefray, E. Caron, J. Rouzaud-Cornabas, and C. Toinard. Microarchitecture-aware virtual machine placement under information leakage constraints. In *IEEE 8th International Conference on Cloud Computing (CLOUD), 2015*, pages 588–595, June 2015.

[79] A. Lefray, E. Caron, J. Rouzaud-Cornabas, H. Zhang, A. Bousquet, J. Briffaut, and C. Toinard. Security-aware models for clouds. Poster at the 22th IEEE International Symposium on High Performance Distributed Computing (HPDC), Jun. 2013.

[80] A. Lefray and J. Rouzaud-Cornabas. Formalisation de propriétés de flux d'information avec une logique temporelle du premier ordre pour assurer la sécurité d'une infrastructure de cloud. In *Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS), 2014*, 2014.

[81] F. Lesueur, L. Me, and V. Tong. An efficient distributed pki for structured p2p networks. In *IEEE Ninth International Conference on Peer-to-Peer Computing, 2009. P2P '09.*, pages 1–10, Sept 2009.

[82] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-based modeling language for model-driven security. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002 — The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 426–441. Springer Berlin Heidelberg, 2002.

[83] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association.

[84] M. Maiza and M. S. Radjef. Heuristics for solving the bin-packing problem with conflicts. *Applied Mathematical Sciences*, 5(35):1739–1752, 2011.

[85] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[86] L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

[87] D. V. Milushev. *Reasoning about Hyperproperties*. Scholars' Press, 2014.

[88] M. Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[89] T. Moses et al. Extensible access control markup language (XACML) version 2.0. *Oasis Standard*, 200502, 2005.

[90] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.

[91] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):410–442, 2000.

[92] P. Nguyen, J. Klein, Y. Le Traon, and M. Kramer. A systematic review of model-driven security. In *Software Engineering Conference (APSEC), 2013 20th Asia-Pacific*, volume 1, pages 432–441, Dec 2013.

[93] K. Okamura and Y. Oyama. Load-based covert channels between Xen virtual machines. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, page 173–180, New York, NY, USA, 2010. ACM.

[94] D. G. O'Brien and W. A. Yasnoff. Privacy, confidentiality, and security in information systems of state health agencies. *American Journal of Preventive Medicine*, 16(4):351 – 358, 1999.

[95] D. F. Parkhill. Challenge of the computer utility. 1966.

[96] S. Pearson. Taking account of privacy when designing cloud computing services. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, pages 44–52, Washington, DC, USA, 2009. IEEE Computer Society.

[97] C. Percival. Cache missing for fun and profit. 2005.

[98] A. Pnueli. The temporal logic of programs. In *The 18th Annual Symposium on Foundations of Computer Science, 1977*, pages 46–57, 1977.

[99] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006 MICRO-39*, pages 135–148, Dec 2006.

[100] H. Raj, R. Nathuji, and A. Singh. Resource management for isolation enhanced cloud services. *CCSW '09 Proceedings of the 2009 ACM workshop on Cloud computing security*, page 77, 2009.

[101] L. Ramakrishnan, H. Rehn, J. Alameda, R. Ananthakrishnan, M. Govindaraju, A. Slominski, K. Connelly, W. Von, D. Gannon, R. Bramley, and S. Hampton. An authorization framework for a Grid based component architecture. In M. Parashar, editor, *Grid Computing — GRID 2002*, volume 2536 of *Lecture Notes in Computer Science*, pages 169–180. Springer Berlin Heidelberg, 2002.

[102] E. Reshetova, J. Karhunen, T. Nyman, and N. Asokan. Security of OS-level virtualization technologies. In K. Bernsmed and S. Fischer-Hübner, editors, *Secure IT Systems*, volume 8788 of *Lecture Notes in Computer Science*, pages 77–93. Springer International Publishing, 2014.

[103] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[104] M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *Computer*, 38(5):39–47, May 2005.

[105] J. Rushby. *Noninterference, transitivity, and channel-control security policies*. SRI International, Computer Science Laboratory, 1992.

[106] P. Ryan, J. McLean, J. Millen, and V. Gligor. Non-interference: Who needs it? In *CSFW*, page 0237. IEEE, 2001.

[107] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan 2003.

[108] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[109] P. Saripalli and B. Walters. Quirc: A quantitative impact and risk assessment framework for cloud security. In *IEEE 3rd International Conference on Cloud Computing (CLOUD), 2010*, pages 280–288, July 2010.

[110] G. J. Simmons. The prisoners' problem and the subliminal channel. In *Advances in Cryptology: Proceedings of CRYPTO '83*, pages 51–67. Plenum, 1983.

[111] G. Smith, C. E. Irvine, D. Volpano, et al. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):1–21, 1996.

[112] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, and J. Lepreau. The flask security architecture: System support for diverse policies. In *Proceedings of the Eighth USENIX Security Symposium*, 1999.

[113] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition edition, 2002.

[114] K. Taesoo, M. Peinado, and G. Mainar-Ruiz. System-Level Protection Against Cache-based Side Channel Attacks in the Cloud. In *Proceedings of the 21st Usenix Security Symposium*, USENIX Security'12, pages 1–16, Berkeley, CA, USA, 2012. USENIX Association.

[115] TCSEC. Trusted Computer System Evaluation Criteria. Technical Report DoD 5200.28-STD, Department of Defense, 1985.

[116] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift. Resource-Freeing Attacks: Improve your Cloud Performance (At your Neighbor's Expense). In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 281–292, New York, NY, USA, 2012. ACM.

[117] A. Wailly, M. Lacoste, and H. Debar. Vespa: Multi-layered self-protection for cloud resources. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 155–160, New York, NY, USA, 2012. ACM.

[118] J. Wainer, P. Barthelmess, and A. Kumar. W-RBAC—a workflow security model incorporating controlled overriding of constraints. *International Journal of Cooperative Information Systems*, 12(04):455–485, 2003.

[119] T. Waizenegger, M. Wieland, T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, B. Mitschang, A. Nowak, and S. Wagner. Policy4tosca: A policy-aware cloud service provisioning approach to enable secure cloud computing. In R. Meersman, H. Panetto, T. Dillon, J. Eder, Z. Bellahsene, N. Ritter, P. De Leenheer, and D. Dou, editors, *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*, volume 8185 of *Lecture Notes in Computer Science*, pages 360–376. Springer Berlin Heidelberg, 2013.

[120] Z. Wang and R. Lee. Covert and side channels due to processor architecture. In *22nd Annual Computer Security Applications Conference, ACSAC '06*, pages 473–482, Dec. 2006.

[121] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security module framework. In *Ottawa Linux Symposium*, volume 8032, 2002.

[122] J. Wu, L. Ding, Y. Wang, and W. Han. Identification and evaluation of sharing memory covert timing channel in Xen virtual machines. In *IEEE International Conference on Cloud Computing (CLOUD), 2011*, pages 283–291, July 2011.

[123] Z. Wu, Z. Xu, and H. Wang. Whispers in the hyper-space: high-speed covert channel attacks in the cloud. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[124] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, page 29–40, New York, NY, USA, 2011. ACM.

[125] F. Yan, W. Qiang, Z. Shen, C. Chen, H. Zhang, and D. Zou. Daonity: An experience on enhancing Grid security by trusted computing technology. In L. Yang, H. Jin, J. Ma, and T. Ungerer, editors, *Autonomic and Trusted Computing*, volume 4158 of *Lecture Notes in Computer Science*, pages 227–235. Springer Berlin Heidelberg, 2006.

[126] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM.

[127] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association.

[128] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, May 2010.

[129] T. Zhang, F. Liu, S. Chen, and R. B. Lee. Side channel vulnerability metrics: The promise and the pitfalls. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 2:1–2:8, New York, NY, USA, 2013. ACM.

[130] X. Zhang, N. Wuwong, H. Li, and X. Zhang. Information security risk management framework for the cloud computing environments. In *IEEE 10th International Conference on Computer and Information Technology (CIT), 2010*, pages 1328–1334, June 2010.

[131] Y. Zhang, A. Juels, A. Oprea, and M. Reiter. HomeAlone: Co-residency detection in the cloud via side-channel analysis. In *IEEE Symposium on Security and Privacy (SP), 2011*, pages 313–328, May 2011.

[132] J. Zimmermann, L. Mé, and C. Bidan. An improved reference flow control model for policy-based intrusion detection. In E. Snekkenes and D. Gollmann, editors, *Computer Security – ESORICS 2003*, volume 2808 of *Lecture Notes in Computer Science*, pages 291–308. Springer Berlin Heidelberg, 2003.

[133] D. Zissis and D. Lekkas. Addressing cloud computing security issues. *Future Generation Computer Systems*, 28(3):583–592, Mar. 2012.