View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by HAL-ENS-LYON

Security for Cloud Environment through Information Flow Properties Formalization with a First-Order Temporal Logic

Arnaud Lefray, Jonathan Rouzaud-Cornabas, Jérémy Briffaut, Christian Toinard

To cite this version:

Arnaud Lefray, Jonathan Rouzaud-Cornabas, Jérémy Briffaut, Christian Toinard. Security for Cloud Environment through Information Flow Properties Formalization with a First-Order Temporal Logic. [Research Report] RR-8420, INRIA. 2013, pp.30. <hal-00916882>

HAL Id: hal-00916882

https://hal.inria.fr/hal-00916882

Submitted on 10 Dec 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Formalisation de propriétés de sécurité via une logique temporelle sur les flux d'information

A. Lefray, J. Rouzaud-Cornabas, J. Briffaut, C. Toinard

RESEARCH REPORT
N° 8420
December 2013
Project-Team Avalon

ISSN 0249-6399 ISRN INRIA/RR--8420--FR+ENG

Formalisation de propriétés de sécurité via une logique temporelle sur les flux d'information

A. Lefray*†, J. Rouzaud-Cornabas‡, J. Briffaut†, C. Toinard†

Project-Team Avalon

Research Report n° 8420 — December 2013 — 27 pages

Abstract: The main slowdown of Cloud activity comes from the lack of reliable security. The on-demand security concept aims at delivering and enforcing the client's security requirements. In this paper, we present an approach, Information Flow Past Linear Time Logic (IF-PLTL), to specify how a system can support a large range of security properties. We present in this paper how to control those information flows from lower system events. We give complete details over IF-PLTL syntax and semantics. Furthermore, that logic enables to formalize a large set of security policies. Our approach is exemplified with the Chinese Wall commercial-related policy. Finally, we discuss the extension of IF-PLTL with dynamic relabeling to encompass more realistic situations through the dynamic domains isolation policy.

Key-words: Cloud, Security, IaaS, Protection, Information Flow, Temporal Logic, First Order

* ENS de Lyon, France, Email: FirstName.LastName@ens-lyon.fr
† ENSI de Bourges, France, Email: FirstName.LastName@ensi-bourges.fr
‡ INRIA, France, Email: FirstName.LastName@inria.fr

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Security for Cloud Environment through Information Flow Properties Formalization with a First-Order Temporal Logic

Résumé : La principale cause de ralentissement de l'adoption du Cloud est le manque de sécurité fiable. Le concept de sécurité à la demande est de déployer et d'appliquer les demandes de sécurité d'un client. Dans ce papier, nous présentons une approche, Information Flow Past Linear Time Logic (IF-PLTL), qui permet de spécifier comment un système peut supporter un large ensemble de propriétés de sécurité. Nous présentons dans ce papier comment ces flux d'information peuvent être contrôler en utilisant les événements systèmes de bas niveau. Nous donnons une description compléte de la syntaxe de IF-PLTL ainsi que sa sémantique. De plus, cette logique permet de formaliser un large ensemble de politiques de sécurité. Notre approche est illustrée par la politique de sécurité de la muraille de Chine orienté vers le monde commercial. Finalement, nous montrons comment nous avons étendu notre langage pour supporter la relabélisation dynamique qui permet de supporter la dynamicité inhérante des systèmes. Nous illustrons cette extension par la formalisation d'une propriété de sécurité pour l'isolation dynamique de domaines.

Mots-clés : Cloud, Sécurité, IaaS, Protection, Flux d'information, Logique temporelle, Premier ordre

1 Introduction

The main slowdown of Cloud activity comes from the lack of reliable security [30, 41]. A Cloud environment faces a variety of threats from traditional threats such as Operating System (OS) (at the hypervisor level or at the VM OS level) or network attacks to new threats arising from having shared environments *e.g.,* shared hardware resources at the IaaS level [18], shared (virtual) OS resources at the PaaS level [37], shared applications resources at the SaaS level [19]. An efficient security policy must consider all layers of multitenancy *i.e.,* IaaS, PaaS and SaaS. Bringing security into Clouds is an active research domain witnessed by various academic and industrial on-going projects [1, 33, 43]. We believe a key concept is about delivering *on-demand security.* An user must be able to *specify* what is the required secure behavior of his system. Hence, we need to provide a language that allows the expression of specific security requirements.

However, the process of specifying/configuring a security policy is usually very complex and error-prone, as presented in [16]. 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. The intended behavior of a system *i.e.,* what a system should or should not do, falls into human knowledge. Therefore, after an initial security specification, the system must follow a Mandatory Access Control (MAC) approach where even the *root* user cannot alter the specified security policy [28].

Halpern *et al.* [27] state that 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.

In this paper, we propose a logic well-suited for a multitenant, concurrent and non-deterministic system.

To be of practical use, a formal system must satisfy the following constraints:

1. It must model how information is exchanged, namely direct, transitive and intransitive information flows.
2. It must model multi-domain, concurrent and non-deterministic systems.
3. It must be usable by non-logicians *i.e.,* the IaaS client.

When correctly configured, many security mechanisms proved to be efficient in practice *e.g.,* SELinux, Iptables. Considering that an administrator should not rely on a unique technology, a formal specification of security requirements should be expressive enough to be mapped to the inner logic of those security mechanisms.

The paper is organized as follows. Section 2 discusses the three constraints and motivates our work by reviewing existing security approaches. Section 3 describes how a system can be modeled as information flow traces. Section 4 describes both syntax and semantics of our *Information Flow Past Linear Time Logic* (IF-PLTL). Section 5 exemplifies high-level properties with the *Chinese Wall.* Section 6 extends IF-PLTL with annotations to dynamically force context relabeling and use it to formalize a *dynamic domains isolation* policy. Finally, Section 7 concludes with open questions and perspectives.

2 Motivations

In this section, we motivate our proposal and explain where it is located in the larger field of protection mechanisms and their underlying logics. First, we discuss about the difference between Information Flow Control (IFC) and Access Control (AC). Next, we differentiate two types of IFC: firstly included in the programing models and secondly included in systems. Then we lay the base of IFC logics and show the importance of having a usable and applicable IFC logic. Finally, we give an overview of our approach.

2.1 Information Flow Control vs Access Control

Information Flow Control and Access Control can be seen as different but maybe complementary approaches to security [31]. Access Control makes explicit statements about permissions for a principal to realize a specific action on a resource whereas information flows are implicit. IFC makes explicit statements about permitted information flows whereas permissions for a specific action are implicit. It is not easy to see if an AC policy can leak information to a potential attacker and many studies have been lead in this direction, namely AC policy analysis [8]. In recent work [36], AC policies differentiate permission (*can do*), obligation (*must do*), denial (*cannot do*) and *do not apply*. Such refinement does not exist in IFC where the objective is to understand and control how information flows between entities that is how entities interfer with each others.

Information Flow (IF) properties are naturally temporal. A flow between two entities is either allowed or denied depending on what flows previously occurred. For example, in the property *Alice can read File if Bob has never written in File*, permitting a flow between *Alice* and *File* is determined by the existence of a previous flow between *Bob* and *File*. This is referred in the literature as history-based IFC [4].

2.2 Direct, Transitive and Intransitive Information Flows

An IF policy between two entities can be either *direct*, *transitive* or *intransitive*. Let suppose an IF policy allowing direct flows from entity A to B and from B to C. A *transitive* policy would allow a direct flow from A to C by transitivity. However, an *intransitive* policy would not allow a direct flow from A to C but will allow a flow from A to C passing through B. These intransitive IF policies are hard to specify [39] but are essential. Therefore, we must include them in our formalization.

2.3 Language-based vs System-based IFC

IFC is studied at two levels. Language-based IFC focuses on flows *inside* a program [40], for example, let *secret* and *public* be two variables. The statement *if secret then public := 1* creates an implicit flow between *secret* and *public*. It is an invasive approach as it requires to expand the code with security annotations to tag the different data. System-based IFC focuses on flows *inside* a system *e.g.,* an Operating System (OS); it controls the flow of information between the entities of a system. System-based IFC is not invasive as it does not required a modification of any applications. But it comes with a loss of granularity as it

is not possible to differentiate variables inside a given application. This paper deals with the System-based IFC.

2.4 Static analysis vs Dynamic monitoring

Information flow control systems tackle any environment from static systems to (runtime) reference monitors. In the static case, IF properties are checked against a complete model of the system, generally a non-deterministic automata. This static analysis is done through *model checking* [24]. In the dynamic case, IF policies are checked at runtime, that is whenever an event generates an information flow, the IF reference monitor decides if this flow satisfies or not the IF policy. Our approach is to propose a formal system encompassing both the static and the dynamic case.

2.5 Roles, Levels, Clearances, Groups and Categories

Attaching labels (or tags) to entities of the system facilitates information control. These labels expose meta-data to efficiently define security domains, types, roles, etc. as well as includes the dynamicity from adding/removing entities.

IFC has been first introduced by Denning *et al.* in [22] with a *Multi-Level System* (MLS). These levels are coming from the military domain where an information from the *top-secret* level cannot flow to a *low-clearance* level or only though *declassification*. The partial ordering between levels, represented by a lattice, is quite used in IFC research [34]. It seems not straightforward to apply such strict model to the IaaS Cloud multitenancy where one of the basics is to share resources. Nonetheless, *Multi-Domain Systems* (MDS) [45] has been proposed to tackle more flexible organizations and is more suitable for Clouds. In this paper, we propose a non-hierarchical organization where entities of a system can belong to multiple *sets* and it is up to the IaaS client to semantically define these *sets* as roles, levels, clearances, groups or categories.

2.6 IFC Logic-based models

AC or IFC systems have been widely modeled with logics. In [27], Halpern applies First-Order (FO) logic to digital rights management. Cassandra [7], a role-based trust management system, specifies AC policies for large-scale systems. Cassandra is based on a logic-programming language, Datalog [29] with constraints. Binder [23] extends Datalog to express distributed security statements. Bruns *et al.* [17] propose a specification in Belnap logic for analyzable AC policy composition. In [5], the authors specify and implement *(Temporal) Role-Based Access Control* policies in *constraint logic programming*.

To our knowledge, the closest work from ours has been proposed by Basin *et al.* in [6]. The authors propose a runtime monitoring with a Metric First-Order Temporal Logic (MFOTL) and provide algorithms to enforce formulas. Their approach is similar to our work in the way that MFOTL includes the (Past)-Linear Time Logic temporal modalities and the system behavior is modeled as (sets of) traces. Nonetheless, the authors' proposal differ from our proposition by not explicitly modeling information flows and by defining a set of relations depending on the use case. We argue that a logic must define

a finite (and fixed) set of relations and functions that can be applied to any system-based IFC.

2.7 IFC Usability and Applicability

Using IFC in Clouds to enforce security requirements has been motivated [3,41] but it has never been applied. In a large scale system like an IaaS Cloud, an IFC system must cautiously consider what information to model and hence monitor (in the dynamic case). In [38], Rushby encodes temporal execution with two functions *step* and *run* which seem equivalent to our temporal modalities, but a third function, *content*, is defined to express the information contained by an object and hence observe changes in contents. To our opinion, it is not feasible to remember the content of objects in a complex system such as an OS, especially at different points in time.

In model-checking approaches, the question is how many states are needed to represent a complex system. In such approaches, an automata-like modelisation is required to represent the system's behavior. This requirement is time-consuming. Nonetheless, it has been applied to operating system policies analysis [14,26]. An other weakness is when the automata changes, all analyses must be redone. Accordingly, due to dynamicity and multitenancy of a Cloud, it is too complex to statically verify security properties.

Usability is also an important factor when proposing new security approaches. Indeed, if describing security requirements is too complex, the security approach will not be used [2]. Accordingly, we propose an high-level API composed of general security policies, each property being formally defined. The IaaS client can pick the required security properties and parameterized it without taking care of formally defining it. Nevertheless, an advanced client will be able to formally define new security properties if required.

The formalization presented in this paper is based on the PIGA language [14, 21] but revisited to be more formal and general. The PIGA language has been used to implement reference monitors in various environments such as:

- PIGA-OS [16], an operating system based on Linux modified to control flows between entities (process, files, etc.).
- PIGA-Virt [15], it controls flows between virtual machines at the hypervisor layer.
- PIGA-Cluster [10], it outsources the protection mechanisms to a dedicated node to avoid performance loss.
- PIGA-Dalvik [11], a SE-Dalvik JAVA monitor controlling flows between Android applications.

PIGA-based systems proved to be efficient, PIGA-OS has won the french national Security Challenge (*aka. "Défi de Sécurité"* from the ANR program SEC&SI) where three security systems and research teams were both protecting an operating system and attacking the others.

2.8 Our approach: Overview

In this section, we have discussed the following constraint *It must model how information is exchanged, namely direct, transitive and intransitive information*

flows and defined the scope of study to system-based IFC for static analysis and dynamic monitoring. The second constraint *It must model multi-domain, concurrent and non-deterministic systems* is motivated first by the Cloud multitenant architecture. The last constraint *It must be usable by non-logicians i.e., the IaaS client* is more intuitive and can be achieved by proposing a simple high-level API.

As shown in Figure 1, the objective is to decide whether the system's behavior modeled as traces *satisfies* a security policy using logic formulas as formal representation. A trace is a sequence of state transitions. A system does not directly produce *information flows* but low-level *observable events* instead. These events are transformed into *functional events* and finally into *Information Flows* (IF). An IF-trace satisfies (or not) a security policy *i.e.,* a set of *closed formulas* called *a theory*. Finally, a user-friendly API is provided as a set of *high-level properties* with parameters, the example of the Chinese Wall policy being given in Section 5 and 6.

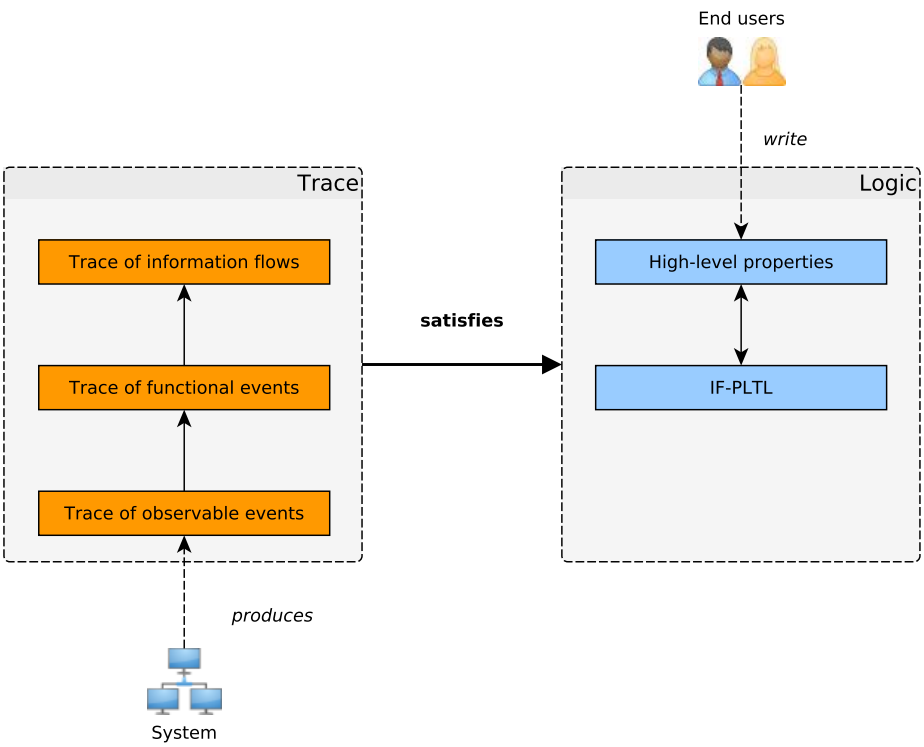

Figure 1: Overview of the satisfaction of a model by logic formulas

3 System Model: Traces

In this section, we present how to construct IF-traces over low-level events.

To define security in an IaaS Cloud composed of virtual and physical OSes, we must represent a concurrent and non-deterministic system. Let 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 (or systems).*

System behavior as traces Every entity inside the Operating System (OS) environment can be viewed as systems *e.g.,* files, processes, databases. Each virtual machine is also a subsystem of the Cloud. A concurrent system behavior can be modeled by all execution traces generated by this system [32], where an execution trace is a sequence of state transitions. We give afterwards a general definition (Definition 4) of traces including the concept of *observer* (Definition 3). Let 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 (Observer) *A passive entity with a system view (potentially partial).*

Definition 4 (Trace) *Sequence of state transitions triggered by events, as viewed by an observer.*

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. In the following, we note $\mathcal{SC}$ the set of all contexts.

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

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

3.1 Traces with Observable Events

The kernel has a complete view of all traces of the operating system. Kernel calls *e.g.,* sys_read, sys_write, sys_fork, are made from an entity to another (potentially newly created in the fork case). In current Linux kernels, the Linux Security Module (LSM) [46] provides all needed kernel hooks. 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 [9].

An *observable event* (Definition 6) is an atomic event viewed by an observer *e.g.,* the kernel.

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

Definition 6 (Observable Event)
$oe \in \mathcal{OE} \equiv_{def} (a, eop, b)$ **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 2 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** $oe_i \in \mathcal{OE}$

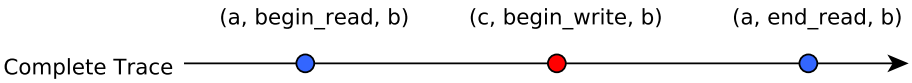

Figure 2: Trace of observable events

3.2 Traces with 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 is able to perform multiple functional operations in parallel. Let 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 with oe_i.

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

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

Definition 8 (Functional Event)
$\forall k \in [i, j], (a, op, b)_k \in \mathcal{FE} \equiv_{def}$
$$\begin{cases} a, b \in SC, op \in \mathcal{FO} \\ \exists i' \leq i, oe_{i'} = (a, eop_{begin}, b)_{i'} \wedge is_begin_event(oe_{i'}, op) \\ \exists j' \geq j, oe_{j'} = (a, eop_{end}, b)_{j'} \wedge is_end_event(oe_{j'}, oe_{i'}) \end{cases}$$

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

Figure 3 is the result of projecting observable events of Figure 2 into functional events. In this trace, the first observable event $(a, begin_read, b)$ and third observable event (a, end_read, b) is transformed into a functional event $(a, read, b)$ occurring at every instant between the beginning and the end included.

Figure 3: Trace of functional events

3.3 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.

In our opinion, the last operator comes in two flavor depending who is issuing the relabeling, an entity or the *observer*. Our definition of an observer (Definition 3) only allows it to evaluate whether or not traces satisfies a given security policy. In this case, the relabeling operation must be initiated by entities, this operation is called *a transition e.g.,* an entity wants to transit from a to b.

We consider the three relations in our information flow traces:

- A flow from a to b is defined as $(a > b)$
- An anti-flow *i.e.,* the absence of flow, from a to b is defined as $(a \not> b)$
- A transition from a to b is defined as $(a >_t b)$

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

is_read_like: $\mathcal{FO} \rightarrow \{true, false\}$
is_write_like: $\mathcal{FO} \rightarrow \{true, false\}$
is_trans_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.
- a transits to b.

Definitions 10, 11, 12 formally define the three relations $(>, \not>, >_t)$ using the functions *is_read_like*, *is_write_like*, *is_trans_like*.

Definition 10 (Flow-to Relation) $(a > b) \in \mathcal{IF} \equiv_{def} \exists op \in \mathcal{FO} \begin{cases} ((b, op, a) \wedge is_read_like(op)) \\ \vee \\ ((a, op, b) \wedge is_write_like(op)) \end{cases}$

Definition 11 (Anti-flow Relation)
$(a \not> b) \equiv_{def} \neg(a > b)$

Definition 12 (Transition Relation)
$(a >_t b) \in \mathcal{IF} \equiv_{def} \exists op \in \mathcal{FO}((a, op, b) \wedge is_trans_like(op))$

Figure 4 is the result of projecting functional events in Figure 3 into information flows. $(a, read, b)$ and $(c, write, b)$ are substituted by $(a < b)$ and $(c > b)$ respectively.

Figure 4: Trace of information flows

3.4 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 three functions *is_read_like*, *is_write_like* and *is_trans_like*, we have described how to finally obtain IF-traces.

The reader should note that despite we used the OS/kernel example, the approach is not limited to this type of systems and has been historically applied to mobile systems (Android), clusters and hypervisors with the PIGA engine [9].

4 Security Properties: Temporal Logic

A set of execution traces of a system is either statically generated from a model *e.g.,* a state machine, a petri net, or dynamically produced at runtime. In the static case, we obtain a set of (potentially) infinite traces representing all possible executions viewed by an observer. In the dynamic case, we obtain a unique finite trace per observer representing the history of the current execution.

A security property is expressed as a property on traces. The same security property must be satisfiable in both the static and the dynamic case. Let takes an example of a trace property, *the reachability problem.* It is defined as follows: *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 but not in the dynamic case. In fact in the second, the trace only represents what *has* occurred but never what *will* occur. Therefore, the answer is *no* until the state is reached. Generally, any property that requires to know the existence of a future state is not satisfiable for dynamic traces. Indeed, with dynamic traces, it is not possible to know in advance the next state of the system.

Definition 13 (Trace Property) *A set of infinite traces.*

Definition 14 (Trace Hyperproperty) *A set of trace properties* i.e., *a set of sets of traces.*

We consider the same definitions (Definition 13 and 14) of properties on traces as defined by Clarkson *et al.* in [20]. The trace property is expressed on traces that comes from a single observer whereas a trace hyperproperty is expressed on sets of traces that come from multiple observers.

4.1 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. A *temporal* logic implicitly defines the flow of time over which formulas are evaluated. A *first-order* logic allows to use quantifiers to express properties such as: *There is a context a from which flows are initiated.* And finally, a *many-sorted* logic, as opposed to a *single-sorted logic*, allows to define several *sorts* of domains instead of an homogeneous domain of

discourse over which a quantifier iterates. Using sorts allows us to easily distinguish *contexts* and *domains* without requiring to use a second-order logic. 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 describe a many-sorted signature (Definition 15).

Definition 15 (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* $n \geq 0$ *and* c_i *is a constant symbol of sort* $\sigma \in S$.
- $F = \{f_1, \ldots, f_n\}$ *where* $n \geq 0$ *and* f_i *is a function symbol of arity* $m \geq 0$ *with sorts* $(\sigma_1 \times \ldots \times \sigma_m) \rightarrow \sigma$ *where* $\sigma_i \in S$ *and* $\sigma \in S$.
- $P = \{p_1, \ldots, p_n\}$ *where* $n \geq 0$ *and* p_i *is a predicate symbol of arity* $m \geq 0$ *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 [12, 35]. (P-)LTL is a well-known formalism to express safety and liveness properties in concurrent systems. Clarkson *et al.* stated that a security property is an intersection between a safety and a liveness property. As a many-sorted logic is well-suited to specify IF properties, (P-)LTL is well-suited to express *temporal* IF properties.

PLTL modalities are the following:

1. $X(\phi)$ **(Next)**: ϕ has to hold at the next state.
2. $G(\phi)$ **(Globally in the future)**: ϕ has to hold on the entire subsequent path.
3. $F(\phi)$ **(Eventually in the future)**: ϕ eventually has to hold (somewhere on the subsequent path).
4. $(\phi)U(\varphi)$ **(Until)**: ϕ has to hold until φ holds.
5. $Y(\phi)$ **(Previous)**: ϕ had to hold at the previous state.
6. $H(\phi)$ **(Globally in the past)**: ϕ had to hold on the entire subsequent path.
7. $P(\phi)$ **(Eventually in the past)**: ϕ eventually had to hold (somewhere on the subsequent path).
8. $(\phi)S(\varphi)$ **(Since)**: ϕ had to hold since φ held.

4.2 IF-PLTL Syntax

The syntax describes how to construct *well-formed* formulas of our logic. A formulae 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.

We first give the signature for temporal information flows and then describe the formation rules.

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

$S = \{ctx, dom\}$

$C = \emptyset$

$F = \emptyset$

$$P = \begin{array}{ll} \{>, \gg, \not>, >_t & : ctx \times ctx \\ \in, \notin & : ctx \times domain \\ \in, \notin & : domain \times domain\} \end{array}$$

We consider two sorts *ctx* and *dom* to designate *contexts* and *domains* *i.e.,* sets of contexts. Similarly to IF-traces, IF relations $(>, \not>, >_t)$ are naturally defined between contexts. We also introduce the *indirect flow* relation $(\gg)$ semantically distinguished from a direct flow $(>)$. This indirect flow does not exist in IF-traces but can be infered from a sequence of direct flows. For example, the IF-trace in Figure 4 satisfies the property $c \gg a$ at the second and third instants because of the indirection $c > b$ and $b > a$.

The set membership relations $(\in, \notin)$ are 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.

Formation rules

Σ-TERM $t ::= x_\sigma \mid c_\sigma \mid f(t_1, \ldots, t_n)$ where

$$\begin{cases} x_\sigma \text{ is a variable of sort } \sigma. \\ c_\sigma \text{ is a constant symbol of sort } \sigma. \\ f : \sigma_1 \times \ldots \times \sigma_n \to \sigma \text{ is a function symbol} \\ \text{ with } \Sigma\text{-TERM } t_i \text{ of sort } \sigma_i. \end{cases}$$

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-FORMULAE $\varphi ::= a \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid$
$\varphi_1 \to \varphi_2 \mid \varphi_1 \leftrightarrow \varphi_2 \mid \forall_\sigma x \varphi(x) \mid \exists_\sigma x \varphi(x) \mid X(\varphi) \mid$
$Y(\varphi) \mid \varphi_1 U \varphi_2 \mid \varphi_1 S \varphi_2$
where a is a Σ-ATOM and x a variable of sort σ

The derived modalities are defined as follows:

$F\varphi \equiv \top U \varphi$
$G\varphi \equiv \neg F \neg \varphi$
$P\varphi \equiv \top S \varphi$
$H\varphi \equiv \neg P \neg \varphi$

We give some abbreviations for simplicity in formulas expression:

$(\forall x \in y)\varphi(x) \equiv (\forall x)(x \in y) \to \varphi(x)$
$(\exists x \in y)\varphi(x) \equiv (\exists x)(x \in y) \wedge \varphi(x)$
$(a \notin s) \equiv \neg(a \in s)$

Well-formed formulae Let give an example of a well-formed formulae:

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

This formulae states that: At every moment k 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. To simplify the notation, we will suppose *sorts* to be implicitly defined. For example, in the formulae $(\forall a, b)(a > b)$, a, b are of sort ctx, the relation $>$ being only defined between contexts.

Deduction rules The purpose of deduction rules is to make proofs and theorems *e.g.,* proving the equivalence between formulas. We dispose of all classic rules of the FO-logic (Hilbert system), of PLTL and Table 1 describes arguments for the four information flow relations ($>$, $\not>$, $>_t$, $\gg$). The simplified notation $(a > b)_i$ describes the relation $>$ between variables a, b (implicitly of sort ctx) at moment i.

Name	**Sequent**	
Intransitive flow	$(a > b)_i \wedge (b > c)_j, i \leq j$	$\vdash (a \gg c)_j$
Anti-flow introduction	$\neg(a > b)_i$	$\vdash (a \not> b)_i$
Anti-flow elimination	$(a \not> b)_i$	$\vdash \neg(a > b)_i$
Transition flow implication	$(a >_t b)_i$	$\vdash (a > b)_i$
Transitive transition flow	$(a >_t b)_i \wedge (b >_t c)_i$	$\vdash (a >_t c)_i$

Table 1: Derived arguments forms

4.3 IF-PLTL Semantics

The semantic describes how to evaluate any well-formed formulae of IF-PLTL. First, we define a FO many-sorted structure (Definition 16) obtained at every moment of the execution; there is no temporal notions in such structure. Next, we define a FO temporal structure (Definition 17) interpreting the flow of time. Then, we give the definition of the satisfaction relation ($\models$) between an IF-PLTL structure and an IF-PLTL formulae. Finally, we exemplify what an interpretation is with a *non-interference* property.

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 formulae (without temporal modalities).

Definition 16 (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. $\forall \sigma_1, \sigma_2 \in S, \mathcal{D}_{\sigma_1} \cap \mathcal{D}_{\sigma_2} = \emptyset$.*

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 17 (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.* $\forall \sigma_1, \sigma_2 \in S, \mathcal{D}_{\sigma_1} \cap \mathcal{D}_{\sigma_2} = \emptyset$.

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:

$$
\begin{array}{ll}
(\mathfrak{M}, k) \models (a > b) & \text{iff } (a > b) \in \mathcal{I}^k(>) \\
(\mathfrak{M}, k) \models (a >_t b) & \text{iff } (a >_t b) \in \mathcal{I}^k(>_t) \\
(\mathfrak{M}, k) \models (a \in s) & \text{iff } (a \in s) \in \mathcal{I}^k(\in) \\
(\mathfrak{M}, k) \models (a \not> b) & \text{iff } (\mathfrak{M}, k) \models \neg(a > b) \\
(\mathfrak{M}, k) \models (a \gg b) & \text{iff } \exists i, j, (0 \leq i \leq j \leq k), \exists_{ctx} c, \\
 & \begin{cases} (\mathfrak{M}, j) \models (c > b) \\ \text{ and} \\ (\mathfrak{M}, i) \models (a > c) \vee (a \gg c) \end{cases} \\
(\mathfrak{M}, k) \models \neg\varphi & \text{iff } (\mathfrak{M}, k) \not\models \varphi \\
(\mathfrak{M}, k) \models \varphi_1 \wedge \varphi_2 & \text{iff } (\mathfrak{M}, k) \models \varphi_1 \text{ and } (\mathfrak{M}, k) \models \varphi_2 \\
(\mathfrak{M}, k) \models \varphi_1 \vee \varphi_2 & \text{iff } (\mathfrak{M}, k) \models \varphi_1 \text{ or } (\mathfrak{M}, k) \models \varphi_2 \\
(\mathfrak{M}, k) \models \varphi_1 \to \varphi_2 & \text{iff } (\mathfrak{M}, k) \models \neg\varphi_1 \text{ or } (\mathfrak{M}, k) \models \varphi_2 \\
(\mathfrak{M}, k) \models \varphi_1 \leftrightarrow \varphi_2 & \text{iff } (\mathfrak{M}, k) \models \varphi_1 \to \varphi_2 \\
 & \text{and } (\mathfrak{M}, k) \models \varphi_2 \to \varphi_1 \\
(\mathfrak{M}, k) \models \forall_\sigma x \varphi(x) & \text{iff } (\mathfrak{M}, k) \models \varphi[x^{\mathcal{I}}/x] \\
 & \text{for all } x^{\mathcal{I}} \in \mathcal{D}_\sigma \\
(\mathfrak{M}, k) \models \exists_\sigma x \varphi(x) & \text{iff } (\mathfrak{M}, k) \models \varphi[x^{\mathcal{I}}/x] \\
 & \text{for some } x^{\mathcal{I}} \in \mathcal{D}_\sigma \\
(\mathfrak{M}, k) \models X\varphi & \text{iff } k + 1 < |T| \text{ and } (\mathfrak{M}, k+1) \models \varphi \\
(\mathfrak{M}, k) \models Y\varphi & \text{iff } 0 < k \text{ and } (\mathfrak{M}, k-1) \models \varphi
\end{array}
$$

$$
\begin{array}{ll}
(\mathfrak{M}, k) \models \varphi_1 U \varphi_2 & \text{iff } \exists i, (k \leq i < |T|), \\
& \begin{cases} (\mathfrak{M}, i) \models \varphi_2 \\ \text{ and} \\ \forall j, (k \leq j < i), (\mathfrak{M}, j) \models \varphi_1 \end{cases} \\
(\mathfrak{M}, k) \models \varphi_1 S \varphi_2 & \text{iff } \exists i, (0 \leq i \leq k), \\
& \begin{cases} (\mathfrak{M}, i) \models \varphi_2 \\ \text{ and} \\ \forall j, (i < j \leq k), (\mathfrak{M}, j) \models \varphi_1 \end{cases}
\end{array}
$$

Satisfiability As previously explained, the existence of a future state is unsatisfiable for a dynamic trace (constructed as events occur). Therefore, using the modalities F, X, U, S may leads to unsatisfiability. Accordingly, we will prefer their past counter-part.

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

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

Let 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.

Instant k	**1**	**2**	**3**	**4**	**5**
Trace	$(a > b)$	$(f > e)$	$(b > f)$	$(f > d)$	$(c >_t f)$
$\mathcal{I}^k(>)$	$\{(a > b)\}$	$\{(f > e)\}$	$\{(b > f)\}$	$\{(f > d)\}$	$\{(c > f)\}$
$\mathcal{I}^k(>_t)$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{(c >_t f)\}$
$\mathcal{I}^k(\in)$	$D_1 = \{a, b, c\}$ $D_2 = \{d, e\}$ $D_3 = \{f\}$	idem	idem	idem	idem
$D_1 : \|D_2$	**true**	**true**	**true**	**false**	**true**

Table 2: Interpretation and satisfaction of the non-interference between D_1 and D_2.

Isolation and Non-interference An important problem in Clouds is the isolation between tenants. Goguens and Meyers [25] define the isolation property based on two *non-interference* properties. The authors denote by $D_1 : |D_2$ that a group D_1 of users does not interfere with a group D_2 of users and the isolation between D_1 and D_2 is defined as a mutual non-interference *i.e.,* $D_1 : |D_2$ and $D_2 : |D_1$. In terms of information flows, the non-interference property $D_1 : |D_2$ means that information cannot directly or indirectly flow from D_1 to D_2 which can be translated as $\forall u_1 \in D_1, \forall u_2 \in D_2, \neg((u_1 \gg u2) \vee (u_1 > u_2))$.

Let show an example of interpretation with the non-interference property. Let suppose the scenario shown in Figure 5. The figure should be read as follows: At time 1, information flow from a to b, then at time 2, information flow from f to e and similarly for times 3 and 4. At time 5, there is a transition from c to f meaning the entity labeled as c wants to be relabeled as f. Table 5 details for any moment k the interpretation of each relation $(>, >_t, \in)$.

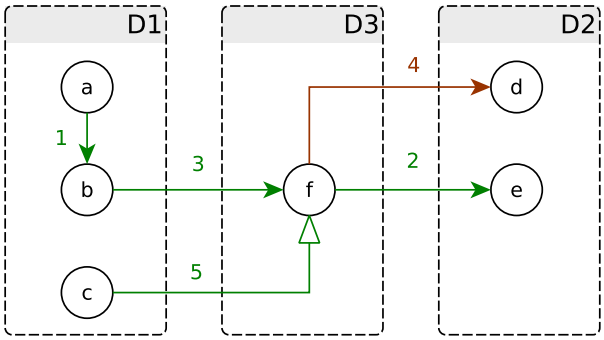

Figure 5: IF-Scenario with three groups (D_1, D_2, D_3) and six entities (a, b, c, d, e, f)

We have motivated the use of a temporal many-sorted logic with information flows and introduced the IF-PLTL syntax and semantics. Moreover, we have shown how to specify a non-interference property in IF-PLTL and verify it against an IF-trace.

5 High-level Security Properties

In this section, we formalize the *Chinese Wall* policy in IF-PLTL to be usable as an high-level API function.

With IF-PLTL, an administrator (or IaaS client) can specify a wide range of IF security properties. For example, Property 1 is a simple property stating that *a formulae occurs at most once during the execution.* The property is constructed as follows: for all moments of the execution (G), if formulae φ is satisfied then φ is not satisfied at any previous moment (YP).

Property 1 (At most one occurence)
$1_{MAX}(\varphi) \Leftrightarrow G(\varphi \rightarrow YP(\neg\varphi))$

Our formalization allows to specify more complex properties. The *Chinese Wall* policy was first introduced by Brewer and Nash [13] and more recently revisited to be less restrictive [42, 44]. In the following, we only discuss Brewer and Nash model.

In the commercial world, a market analyst working for a financial institution is provided with confidential information from his firm's clients. But advising concurrent corporations with insider knowledge creates a conflict of interest and must be forbidden. Therefore, such analyst is free to advise corporations which are not in competition. In the initial model, the authors distinguish *subjects* *e.g.,* a market analyst and *objects* *e.g.,* companies information. As shown in Figure 6, corporation information are stored in hierarchical sets. Individual objects are grouped into company datasets and each dataset belongs to a conflict of interest class. For example, suppose three companies Bank-1, Bank-2 and Telecom-1; there is two classes of interest. Bank-1 and Bank-2 datasets belong to the banking class of interest and Telecom-1 dataset belong to the telecom class of interest. An analyst cannot access both Bank-1 and Bank-2 datasets but can freely access Telecom-1 datasets.

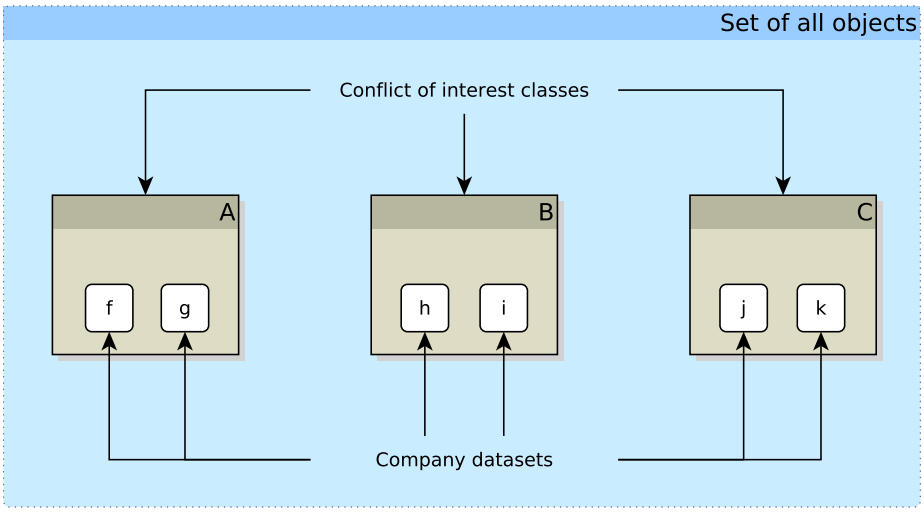

Figure 6: Composition of objects in the Chinese Wall policy

The Chinese Wall policy is axiomatized with four properties:

1. If any object o_1 and o_2 belongs to the same company dataset then they also belong to the same conflict of interest class.

2. Access to any object o_r by a subject s is granted if and only if for all previously accessed objects o_c (by s), o_r and o_c are in the same company dataset or in different conflict of interest classes.

3. No previous access to any objects by any subjects is an initially secure state.

4. Any access by a subject s is granted if he has never accessed any objects before.

To represent this policy in IF-PLTL, we define four sets of entities:

- S, the set of subjects.
- O, the set of objects.
- CDs, the set of company datasets.
- $COIs$, the set of conflict of interest classes.

Then we translate the four Chinese Wall policy axioms into our IF-PLTL language respectively in Axiom 1,2,3,4. And the Chinese Wall is formalized as a trace property in the Property 2.

Axiom 1 $Axiom_1(S, O, CDs, COIs) \Leftrightarrow$
$G\big((\forall o_1, o_2 \in O)(\exists CD_i \in CDs)(o_1, o_2 \in CD_i) \rightarrow$
$(\exists COI_j \in COIs)(o_1, o_2 \in COI_j)\big)$

Axiom 2 $Axiom_2(S, O, CDs, COIs) \Leftrightarrow$
$(\forall o \in O)(\forall s \in S)$
$G\big(((s > o) \vee (o > s)) \rightarrow YH($
$(\forall o' \in O)((s > o') \vee (o' > s)) \rightarrow$
$((\exists CD_i \in CDs)(o, o' \in CD_i)$
$\vee (\forall COI_j \in COIs)(o, o' \notin COI_j))$
$))$

Axiom 3 *In the initial statement of the Chinese Wall policy, the authors model accesses with an access matrix. This axiom states that for every couple (subject, object), having the access matrix initialized at $false$ is a secure state. In our model, an access must occur at a moment k during the execution. In particular, the existence of an access in an empty trace is false. As a result, this axiom is already implicit with our structure.*

Axiom 4 $Axiom_4(S, O, CDs, COIs) \Leftrightarrow$
$(\forall o \in O)(\forall s \in S)G\big($
$((s > o) \vee (o > s)) \rightarrow (\forall o' \in O)YH((s \not> o') \wedge (o' \not> s))\big)$

Property 2 (Chinese-Wall) $CWSP(S, O, CDs, COIs)$
$\Leftrightarrow Axiom_1 \wedge (Axiom_2 \vee Axiom_4)$

Consequently, an IaaS client just has to call the high-level API function $CWSP$ with user-defined parameters $(S, O, CDs, COIs)$. For example:

$S = \{ctx_analyst\}$
$O = \{ctx_data_bank1, ctx_data_bank2, ctx_data_telecom1\}$

$CDs = \{CD_Bank1, CD_Bank2, CD_Telecom1\}$

$CD_Bank1 = \{ctx_data_bank1\}$

$CD_Bank2 = \{ctx_data_bank2\}$

$CD_Telecom1 = \{ctx_data_telecom1\}$

$COIs = \{COI_Bank, COI_Telecom\}$

$COI_Bank = \{CD_Bank1, CD_Bank2\}$

$COI_Telecom = \{CD_Telecom1\}$

6 Dynamicity: Sets Operators

As illustrated by the non-interference example (Figure 5 and Table 5), the interpretation of sets relation ($\in$) is identical at any moment k. Moreover, there is no function nor relation allowing to change this interpretation during the execution *i.e.,* modifying sets membership. In the following, we present a policy, *domains isolation*, where dynamic sets could be of great interests and then detail our proposition based on annotations in formulas and finally describe the *dynamic domains isolation* policy. Indeed, dynamic sets are necessary to increase the flexibility of property and being able to create properties that take into account the dynamicity of systems.

Let suppose a strict domains isolation policy defined by Property 3 where two entities (a, b) can only exchange information if they are in the same domain (Dom_i). With such policy, an entity which does not belong to any domains cannot send any information at all. Therefore, all entities must be properly labelled with a domain which is a complex and time-consuming task.

Property 3 (Domains Isolation)
$DI(DOMs) \Leftrightarrow (\forall a, b)G((a > b) \rightarrow (\exists Dom_i \in DOMs)(a, b \in Dom_i))$

Let consider the scenario depicted in Figure 7. A company outources its services in the Cloud. This company has two departments, R&D and human resources (HR), and each department has confidential data. The R&D dept. has a project manager database (*project_data*). The HR dept. has all employees private information (*employees_data*). Because of the R&D branch, the company has to test unstable releases of a software (*unstable_app*) which is unsafe and must be isolated from other services. Also, the company is sometimes required to test external programs they do not trust (*untrusted_app*). Finally, the company dispose of several applications ($company_app1, company_app2$) that may change over time *e.g.,* adding $company_app3$. The company policy states that an application cannot access both R&D and HR domains but it does not have any *a priori* knowledge of whom the application is for. The idea is to contaminate applications with the domain they belong to at the first access *e.g.,* including $company_app1$ in R&D if it tries to access *project_data*.

Our dynamic domains isolation policy is specified with three properties. Let suppose $DOMs$ to contain domains and sanboxes, then information can flow from a context a to a context b if :

1. a, b are in the same set Dom_i (in $DOMs$). This is the static domains isolation (Property 3).

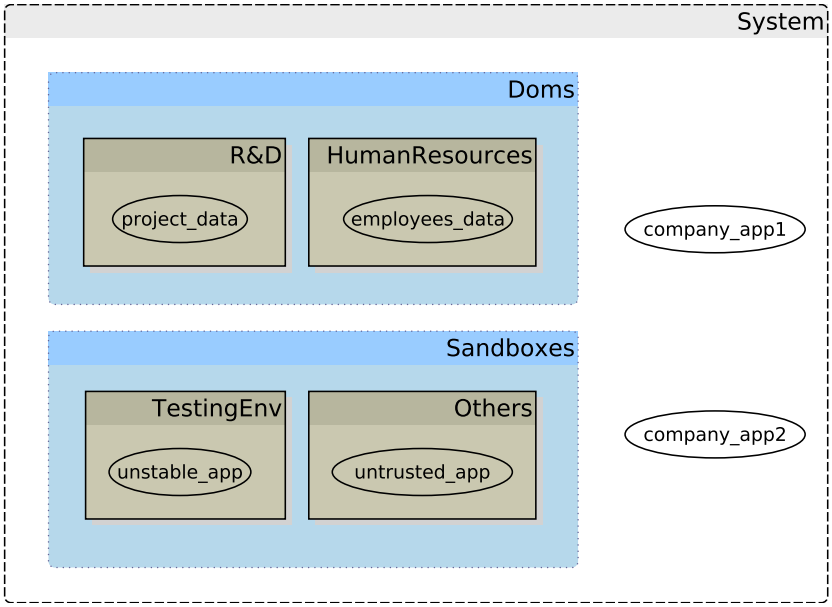

Figure 7: Scenario for dynamic domains isolation with different department services.

Table 3: Dynamic Domains Isolation rules for $(a > b)$

b \ a	Dom_i	Dom_j	$Sandbox_k$	$Sandbox_l$	$None$
Dom_i	$\top$				$\top$
Dom_j		$\top$			$\top$
$Sandbox_k$			$\top$		$\top$
$Sandbox_l$				$\top$	$\top$
$None$	$\top[\![Dom_i]\!]$	$\top[\![Dom_j]\!]$	$\top[\![Sandbox_k]\!]$	$\top[\![Sandbox_l]\!]$	$\top$

2. **or** a is not in any sets.

3. **or** a is in a set Dom_i and b is not in any sets, then b is now member of Dom_i.

We propose the syntax $\varphi[\![Annotation]\!]$ meaning, *Annotation* is realized when φ is true. An annotation to change sets membership would be $[\![S \leftarrow S \bigcup \{x\}]\!]$ that is x becomes a member of S.

In Table 3, we detail restrictions on flows depending on the source (a) / destination (b) and dynamic operations to realize. By $\top$, we denote an allowed flow (denied when empty). The notation $\top[\![Dom_i]\!]$ means that the flow from a to b is allowed and from now on, context b belongs to domain Dom_i. Three different sets are distinguished, namely **Doms** (R&D, HumanResources), **Sandboxes** (TestingEnv, Others) and **None** including every context neither in any Doms nor in any Sandboxes *e.g., company_app*1. Property 4 is the translation of the three properties in IF-PLTL.

Property 4 (Dynamic Domains Isolation)
$DDI(DOMs) \Leftrightarrow$
$$\begin{cases} DI(DOMs) \\ \vee(\forall a,b)G((a>b) \rightarrow (\forall Dom_i \in DOMs)(a \notin Dom_i)) \\ \vee(\forall a,b)(\exists Dom_i \in DOMs)G\big(((a>b) \rightarrow (a \in Dom_i) \\ \wedge(\forall Dom_j \in DOMs)(b \notin Dom_j))[\![Dom_i \leftarrow Dom_i \bigcup \{b\}]\!]\big) \end{cases}$$

We have shown the benefits of using annotations to extend our logic with dynamic relabelling. Furthermore, annotations can be used for specifying *active* capabilities to the (no longer) *passive* observer.

7 Conclusion

First, we have motivated the need of a formal IFC language for distributed and parallel systems and show the difference between our approach and previous ones. Then we have presented our formal language that allows to express advanced security properties related to the control of different types of information flow. Our approach distinguishes direct, transitive, and intransitive flows. Using those operators our Information Flow Past Linear Time Logic (IF-PLTL) enables to express various security properties or policies such as the noninterference or the Chinese Wall. Furthermore, we have shown how IF-trace can be constructed from low-level events happening on a system. To ease the usage of IF-PLTL, we have proposed a high-level API that can be used and parameterized by IaaS client to express their security requirements. Finally, we have shown how we have extended IF-PLTL to take into account the dynamicity of systems and propose a new security policy called Dynamic Domains Isolation. Despite that the underlying idea has already been used in various systems like PIGA-OS and PIGA-Virt, it is the first time that the PIGA language is transposed into a formal logic approach.

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.

References

[1] Cloud Security Alliance. https://cloudsecurityalliance.org/.

[2] Smack. http://schaufler-ca.com/.

[3] J. Bacon, D. Evans, D. M. Eyers, M. Migliavacca, P. Pietzuch, and B. Shand. Enforcing End-to-End Application Security in the Cloud (Big Ideas Paper). In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware '10, pages 293–312, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] A. Banerjee and D. A. Naumann. History-based access control and secure information flow. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, number 3362 in Lecture Notes in Computer Science, pages 27–48. Springer Berlin Heidelberg, Jan. 2005.

[5] 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.

[6] 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.

[7] 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, 2004. POLICY 2004.*, pages 159–168, 2004.

[8] 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.

[9] 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.

[10] M. Blanc, D. Gros, J. Briffaut, and C. Toinard. Mandatory access control with a multi-level reference monitor: PIGA-cluster. In *ACM CLHS '13 Proceedings of the first workshop on Changing landscapes in HPC security*, pages 1–8, New-York, USA, June 2013. ACM.

[11] A. Bousquet, J. Briffaut, L. Clévy, C. Toinard, and B. Venelle. Mandatory Access Control for the Android Dalvik Virtual Machine. In *2013 - USENIX Federated Conferences, ESOS: Workshop on Embedded Self-Organizing Systems*, San Jose, États-Unis, June 2013.

[12] 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.

[13] D. Brewer and M. Nash. The chinese wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy, 1989.*, pages 206–214, 1989.

[14] 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, Dec. 2009. ISSN: 1942-2636.

[15] J. Briffaut, J. Rouzaud-Cornabas, C. Toinard, and E. Lefebvre. PIGA-Virt: an Advanced Distributed MAC Protection of Virtual Systems. In *Euro-Par 2011 Parallel Processing Workshops, Lecture Notes in Computer Science*, pages 8–19, Bordeaux, France, Aug. 2011.

[16] J. Briffaut, J. Rouzaud-Cornabas, C. Toinard, and Y. Zemali. A new approach to enforce the security properties of a clustered high-interaction honeypot. In *High Performance Computing Simulation, 2009. HPCS '09. International Conference on*, pages 184–192, 2009.

[17] 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.

[18] 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.

[19] R.-A. Cherrueau, M. Südholt, and O. Chebaro. Adapting workflows using generic schemas: application to the security of business processes. In *IEEE CloudCom 2013*, Bristol, UK, Dec. 2013.

[20] M. R. Clarkson and F. B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, Sept. 2010.

[21] P. Clemente, J. Rouzaud-Cornabas, and C. Toinard. From a generic framework for expressing integrity properties to a dynamic mac enforcement for operating systems. In M. Gavrilova, C. Tan, and E. Moreno, editors, *Transactions on Computational Science XI*, volume 6480 of *Lecture Notes in Computer Science*, pages 131–161. Springer Berlin Heidelberg, 2010.

[22] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[23] J. DeTreville. Binder, a logic-based security language. In *Proceedings of the IEEE Symposium on Security and Privacy, 2002.*, pages 105 – 113, 2002.

[24] R. Dimitrova, B. Finkbeiner, M. Kovács, M. N. Rabe, and H. Seidl. Model checking information flow in reactive systems. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, number 7148 in Lecture Notes in Computer Science, pages 169–185. Springer Berlin Heidelberg, Jan. 2012.

[25] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and privacy*, volume 12, 1982.

[26] 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.

[27] 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.

[28] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, Aug. 1976.

[29] 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*, page 1213–1216, 2011.

[30] T. Jaeger and J. Schiffman. Outlook: Cloudy with a Chance of Security Challenges and Improvements. *IEEE Security & Privacy Magazine*, 8(1):77–80, Jan. 2010.

[31] 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.

[32] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, Apr. 1983.

[33] B. Marquet, G.-B. Kamga, S. Betgé-Brezetz, M.-P. Dupont, L. Clevy, B. Venelle, A. Bousquet, J. Briffaut, C. Toinard, J.-M. Lambert, E. Caron, A. Lefray, L. Toch, and J. Rouzaud-Cornabas. *Seeding the Cloud: An Innovative Approach to Grow Trust in Cloud Based Infrastructures*, volume The Future Internet Assembly (FIA). FIA Book 2013 of *Lecture Notes in Computer Science (LNCS).* FIA Book 2013 Publisher, Springer edition, 2013.

[34] B. Montagu, B. Pierce, R. Pollack, and A. Surée. A theory of information-flow labels. *Draft, July*, 2012.

[35] A. Pnueli. The temporal logic of programs. In *the 18th Annual Symposium on Foundations of Computer Science, 1977*, pages 46–57, 1977.

[36] C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes. SPL: an access control language for security policies with complex constraints. In *Proceedings of the Network and Distributed System Security Symposium*, 2001.

[37] L. Rodero-Merino, L. M. Vaquero, E. Caron, F. Desprez, and A. Muresan. Building safe paas clouds: a survey on security in multitenant software platforms. *Computers & Security*, 31:96–108, 2012.

[38] J. Rushby. *Noninterference, transitivity, and channel-control security policies.* SRI International, Computer Science Laboratory, 1992.

[39] J. Rushby. Security requirements specifications: How and what. In *Symposium on Requirements Engineering for Information Security (SREIS)*, volume 441, 2001.

[40] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Perspectives of Systems Informatics*, page 352–365. Springer, 2010.

[41] R. Sandhu, R. Boppana, R. Krishnan, J. Reich, T. Wolff, and J. Zachry. Towards a Discipline of Mission-Aware Cloud Computing. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, pages 13–18, 2010.

[42] R. S. Sandhu. Lattice-based enforcement of chinese walls. *Computers & Security*, 11(8):753–763, Dec. 1992.

[43] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[44] A. Sharifi and M. V. Tripunitara. Least-restrictive enforcement of the chinese wall security policy. In *Proceedings of the 18th ACM Symposium on Access Control Models and Technologies*, SACMAT '13, page 61–72, New York, NY, USA, 2013. ACM.

[45] W. She, I.-L. Yen, B. Thuraisingham, and E. Bertino. The SCIFC model for information flow control in web service composition. In *IEEE International Conference on Web Services, 2009. ICWS 2009*, pages 1–8, 2009.

[46] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security module framework. In *Ottawa Linux Symposium*, volume 8032, 2002.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399