5

Non-Interference in Language-based Security

When we imagine, we can only see,
when we know we can compare.
Jean-Jacques Rousseau

In the last decades, an important task of language based security is to protect confidentiality of data manipulated by computational systems. Namely, it is important to guarantee that no information, about confidential/private data, can be caught by an external viewer. In many fields, where protection of confidentiality is a critical problem, the standard way used to protect private data is access control: special privileges are required in order to read confidential data. Unfortunately, these methods allow to restrict accesses to data but cannot control propagation of information. Namely once the information is released from its container, it can be improperly transmitted without any further control. This means that the security mechanisms, such as signature, verification, and antivirus scanning, do not provide assurance that confidentiality is maintained during the whole execution of the checked program. This implies that, to ensure that confidentiality policies are satisfied, it becomes necessary to analyze how information flows within the executed program. In particular, if a user wishes to keep some data confidential, he might state a policy stipulating that no data visible to other users is affected by modifying confidential data. This policy allows programs to manipulate and modify private data, as long as visible outputs of those programs do not reveal information about these data. A policy of this sort is called *non-interference* policy [68], since it states that confidential data may *not interfere* with public data. Non-interference is also referred as *secrecy* [111], since confidential data are considered *private*, while all other data are public [39]. The difficulty of preventing a program P from leaking private information depends greatly on what kind of observations of P are possible [109]. If we can make *external observations* of P's running time, memory usage, and so on, then preventing leaks becomes very difficult. For example, P could modulate its running time in order to encode the private information. Furthermore, these modulations might depend on low level implementation details, such as caching behaviours. But this means that it is insufficient to prove confinement with respect to an abstract semantics, every implementation detail, that affects running time, must be addressed in the proof of confinement. If, instead, we can only make *internal observations* of P's behaviour, the confinement problem become more tractable [109]. Internal observations include the values of program variables, and everything is observable internally, e.g. time in real-time systems.

In this chapter, we provide an excursus on the different notions of non-interference, in different computer science fields, and we describe the main approaches studied (see [104] for a survey). In the following, we first provide a brief background of the notion of non-interference, from the Lampson's formalization of the *confinement problem* [80] to the Cohen's *strong dependency* [19; 20], to the Goguen and Meseguer's definition of *non-interference* [68]. We conclude this part, introducing the semantic approach to non-interference of Joshi and Leino [78] and the PER model, applied to non-interference by Sabelfeld and Sands [106]. At this point, we provide a background about the existing techniques used for *enforcing* non-interference. Starting from the initial access control methods, such as the Bell and LaPadula model [13], we arrive to introduce the Denning and Denning information flow static analysis [38]. We conclude this part describing the Smith and Volpano security type system [114] and the axiomatic approaches to non-interference [7, 6]. Afterwards, we describe how this notion has been extended in order to cope with richer and more complex computational systems (e.g., non-deterministic and multi-threaded languages, process algebras and timed automata). We also introduce the notion of covert channel and we describe some existing solutions studied for avoiding this kind of information flows (e.g, timing and probabilistic channels, termination channels, and so on). Finally, we describe some existing weakenings of the notion of non interference, from the quantitative approaches that measures the information released [17, 84], to the definition of robust declassification [118], from the probabilistic approach characterizing how much statistical tests are necessary to disclose secrets [41], to the complexity-based approach which determines how complex is to disclose secrets [82].

5.1 Background: Defining non-interference

We have underlined above, how the problem of keeping confidential data private can be modeled as a non-interference problem, by stating that secure programs can manipulate and modify private data, as long as visible outputs of those programs do not reveal confidential information. In this section, we describe how non-interference can be defined in different fields of computer science, depending on what the low level user is supposed to be able to observe. Before entering in the specific of the non-interference notion, we want to define what is a *security property*. Consider a set SC of *security classes* [38] (also called security domains in [86]), corresponding to disjoint classes of information. Suppose that each object e of a system is bound to a security class in SC, denoted $dom(e)$, which specifies the security class associated with the information represented by e. In general, a security domain can be, e.g., a group of users, a collection of files or a memory section. A *security property* is composed of a *non-interference relation* $\not\rightsquigarrow \subseteq \mathrm{SC} \times \mathrm{SC}$, which formalizes a *security policy* by stating which domains may not interfere with others, with a *definition of security* [86]. In the following, we simplify and consider only two domains, private/high `H` and public/low `L`, and the security policy which demands that `H` must not interfere with `L`, i.e., `H` $\not\rightsquigarrow$ `L`.

In order to describe the background of the notion of *security* as absence of *flows* from private to public we have to go back to the seminal paper [80] where the notion of *confinement problem* is introduced (also known as *secrecy*). Consider a *customer* program and a *service* (host) program, the customer would like to ensure that the service cannot access (read or modify) any of his data, except those information to which he explicitly grants access (said *public*). In other words, the confinement problem consists in *preventing the results of the computation from leaking even partial information about confidential inputs.* Clearly, if the public data depends, in any way, on the private ones, then confinement becomes a problem. This strict relation between the confinement problem and the dependencies among data allows to describe the confinement problem as a problem of *non-interference* [68] by using the notion of *strong dependency* introduced in [19]. In the latter, the transmission of information is defined by saying that *information is transmitted over a channel when variety is conveyed from the source to the destination.* Clearly, if we substitute source with private and destination with public, then we obtain the definition of insecure information flow. More formally speaking, Cohen in [19] says that information can be transmitted from a to b during the execution of a system S, if by suitably varying the initial value of a (exploring the variety in a), the resulting value in b after S's execution will also vary (showing the variety is conveyed to b). The absence of strong dependency has been interpreted as non-interference in [68], where non-interference is defined as:

> *"One group of users [...] is noninterfering with another group of users if what the first group does [...] has no effect on what the second group of users can see".*

Starting from this informal definition, a non-interference policy which states that a group of users G does not interfere with another group of users G' is defined by saying that what any user $u \in G$ can observe when the machine is in the state representing the effect of an input string w on the states, starting from the initial state of the whole system, denoted by $[\![w]\!]_u$, is the same of what it can observe by

erasing all the actions of users in G'.
Therefore, we have that security, defined as presence of only secure information flows, is non-interference, which is absence of strong dependencies. These definitions are general and can be applied to different kind of computational systems, as we will see later on.
The notion of non-interference is used to stipulate policies of non-interference whenever a user wishes to keep some data confidential. This policy allows programs to manipulate and modify private data so long as visible outputs of those programs do not reveal confidential information. Therefore these policies stipulate that no data visible to other users is affected by confidential data [68].

5.1.1 Cohen's strong and selective dependency

Starting from the observation that in sequential programs information can be transmitted among variables, Cohen noted that the approaches previously studied were almost intuitionistic. His aim was that of providing a formal approach to information transmission so that information paths can be determined precisely given the formal semantics of a program. Moreover, the formal approach allows to answer more *selective* questions about information transmission. For example, we may not care if the output variable b reflects whether the input variable a is odd or even. However we might like to show that b depends on a in *no other way*. This leads clearly both to a semantic formalization of the problem and to a way for weakening the problem itself. In information theory, information can be transmitted from a source a to a destination b if a *variety* can be conveyed from a to b, namely as the result of program execution [19, 20]. This is exactly the idea used for defining *strong dependency*. The selective aspect of dependency, called *selective dependency*, comes from the observation that assertions, constraints on inputs of computation, can eliminate certain information paths, for example making a test always true. Cohen considers a simple imperative language with the usual semantics.

The idea for defining strong dependency is that of considering that if the input a may initially take on a number of different values, resulting in a number of different values in b after the execution of the program P, then we can say that b strongly depends on a. To show that information transmission is possible, we need only to find two different input values for a that yield different values for b after the execution of P. Therefore, adapting Cohen's definition to the security framework, we consider $\mathtt{L}$ as the set of public variables and $\mathtt{H}$ as the set of private ones. Then we say that the variables $\mathtt{L}$ are *strongly dependant* on $\mathtt{H}$ in the program P, $\mathtt{H} \rhd^P \mathtt{L}$, namely the program is not secure, if

$$\exists s_1, s_2 \,.\, s_1^{\mathtt{L}} = s_2^{\mathtt{L}} \;\wedge\; [\![P]\!](s_1)^{\mathtt{L}} \neq [\![P]\!](s_2)^{\mathtt{L}}$$

where s_1 and s_2 are states of P, namely tuples of values for the variables in P, and $s^{\mathtt{L}}$ is the tuple of values in s for the variables in $\mathtt{L}$.

Cohen realized that this definition, in some situations, was too strong, moreover he noted that adding input assertions reduces the information that can be transmitted. In general, any addition or strengthening of an input assertion may reduce (and never increase) information transmission [20]. This consideration leads him to the definition of the *selective dependency*: Often we are not interested in the fact that information is indeed transmitted from one object to another as long as specific properties, portions of the information, are protected. Consider the program

$$P \overset{\text{def}}{=} b := x + (a \; mod \; 4)$$

We can note that b does depend on a, i.e., $a \rhd^P b$, but only upon the last two bits of a. We can prove that the rest of a is protected from b by using strong dependency with a constraint, for example $\phi : \; a \; mod \; 4 = 3$. We can note that, even though P conveys variety from a to b, ϕ eliminates all the variety that is conveyed. Therefore, we define selective dependency for security (with a simplified notation). Consider as above $\mathtt{H}$ and $\mathtt{L}$ in a program P. We say that $\mathtt{L}$ is *selectively independent* from $\mathtt{H}$ in P, as regards the assertion ϕ, i.e., $\mathtt{H} \not\rhd^P_\phi \mathtt{L}$, if:

$$\forall s_1, s_2 \,.\, (\phi(s_1^{\mathtt{H}}) \;\wedge\; \phi(s_2^{\mathtt{H}}) \;\wedge\; s_1^{\mathtt{L}} = s_2^{\mathtt{L}}) \;\Rightarrow\; [\![P]\!](s_1)^{\mathtt{L}} = [\![P]\!](s_2)^{\mathtt{L}}$$

The role of ϕ is clearly that of characterizing which information we admit to flow from $\mathtt{H}$ to $\mathtt{L}$, indeed in the previous example any possible constraint on the value $a \; mod \; 4$ makes b selectively independent from a.

5.1.2 Goguen-Meseguer non-interference

In [68] the authors treat directly the problem of information transmission for enforcing program's security. Their approach to non-interference is intended to deal with both the abstract conceptual level of the security problem, where general concepts and methods are described, and the concrete modelling level, where actual systems are modeled in order to prove that they are secure in any sense. The definition introduced by Goguen and Meseguer is based on the notion of *security policy*, which defines the security requirements for a secure system. Therefore, in this context, security verification consists of showing that a given policy is satisfied by a given model. In general, information flow techniques attempt to analyze how users (or processes, or variables) can potentially *interfere* with other users. On the other hand, the security policy wants to say when users (or processes, or variables) must not interfere with other users. The purpose of a so-called *security model* is to provide a basis for determining whether or not a system is secure, and if not, for detecting its flaws.
In [68], the basic definition used to make non-interference precise considers systems as machines having a set of users U, a set of commands (changing-states) C, a set of read commands R, a set of outputs O and a set of internal states S, with initial state s_0. Moreover, there is a *next* function: $\mathbf{do} : S \times U \times C \longrightarrow S$, where

$\mathbf{do}(s, u, c)$ gives the next state after the user u executes the command c in the state s; and an *output* function: $\mathbf{out} : S \times U \times R \longrightarrow O$ where $\mathbf{out}(s, u, r)$ gives the result of a user u executing a read command r in the state s. Therefore, output commands have no effect on states, and state commands produce no output. The *history* of a system is the sequence $w = \langle (u_1, c_1) \dots (u_n, c_n) \rangle$ where all the pairs are of commands $c_i \in C \cup R$ with their users $u_i \in U$, since the initial startup of the system is in the state s_0. When reasoning about states we can omit all output commands from the history, since output commands do not affect states. Thus, the state reached after the execution, in the system, of a sequence of state commands, starting from the initial state s_0, is given by the function $\mathbf{do}^* : S \times (U \times C)^*$ defined inductively by $\mathbf{do}^*(s, \text{empty}) = s$, $\mathbf{do}^*(s, \langle w(u, c) \rangle) = \mathbf{do}(\mathbf{do}^*(s, w), u, c)$. Let $[\![w]\!]$ denote the state reached from s_0 after the execution of the sequence w, i.e., $[\![w]\!] = \mathbf{do}^*(s_0, w)$.
A non-interference assertion expresses that a certain group G of users executing a certain set $\mathtt{H}$ of state transition commands does not interfere with, i.e., cannot be detected by, another group of users G' executing a set $\mathtt{L}$ of output commands; this is denoted $G, \mathtt{H} :| G', \mathtt{L}$. This assertion holds if and only if for each sequence $w \in (U \times C)^*$, each $v \in G'$, and each $l \in \mathtt{L}$ we have:

$$\mathbf{out}([\![w]\!], v, l) = \mathbf{out}([\![P_{G,\mathtt{H}}(w)]\!], v, l)$$

where $P_{G,\mathtt{H}}(w)$ is the sequence obtained from w by eliminating all occurrences of pairs (u, c) with $u \in G$ and $c \in \mathtt{H}$. Intuitively, this means that whatever any $v \in G'$ can tell by executing output commands in $\mathtt{L}$, everything looks as if the users in G had never executed any commands in $\mathtt{H}$. This is mostly the core work in [68] and it is a slightly different notion from the one introduced by Cohen. Indeed here we require that whenever private actions are executed the output observable behaviour has to be as if no private actions have been executed at all. In sequential programs the private actions are those where private variables are modified, and therefore in general it is a very strong requirement to extract computations where the execution of private actions are avoided. This impose the definition of a weaker notion of non-interference that we will call *standard* non-interference, and which is defined as the negation of Cohen's strong dependency, where private actions may interfere with the output behaviour, unless they do not convey a variety.

5.1.3 Semantic-based security models

As we have seen in the formalization of the Cohen's strong dependency, the problem of non-interference can be characterized by considering semantics of systems. A semantic approach has several features. First, it gives a more precise characterization of security than other approaches. Second, it applies to any programming constructs whose semantics are definable, for example, the introduction of non-determinism poses no additional problems. Third, it can be used for reasoning about indirect leaking of information through variation of the program behaviour (e.g., whether or not the program terminates). Finally, it can be extended to the case where the high and the low security variables are defined abstractly, as function of the actual program variables [78]. We introduce here two main semantic approaches.

A semantic approach to secure information flow.

As we said above, a program is secure if any observation of the initial and final values of the low variables, denoted $l : \mathtt{L}$, do not provide any information about the initial value of the private variables, denoted $h : \mathtt{H}$ [78]. Assume that the adversary has knowledge of the program text and of the initial and final values of l. The idea of Joshi and Leino's semantic-based approach to language-based security is that of characterizing secure information flow as program equivalence, denoted by $\doteq$. They introduce a program $\mathtt{HH} \overset{\text{def}}{=}$ "assign to h an arbitrary value". Consider a program P, for which we want to prove non-interference. The program $\mathtt{HH}; P$ corresponds to run P after having set h to an arbitrary value; while the program $P; \mathtt{HH}$ discards the final value of h resulting from the execution of P. Then a program P is said to be *secure* if

$$\mathtt{HH}\ ;\ P;\ \mathtt{HH}\ \doteq\ P\ ;\ \mathtt{HH} \tag{5.1}$$

where $\doteq$ is the relational input/output semantic equality between programs, namely for each possible input the two programs have to show the same public output behavior. In order to understand this characterization, note that the occurrence of $\mathtt{HH}$ after P on both the sides of the equality indicates that only the final values of l are of interest, whereas the occurrence of $\mathtt{HH}$ before P on the left side of the equality indicates that the program starts with an arbitrary assignment to h. Clearly, the two programs are input/output equivalent provided that the final value of l, produced by P, does not depend on the initial value of h, which is indeed standard non-interference.

PER's model.

The semantic approach described above has also been equivalently formalized by using *partial equivalence relations (PER)* [106]. In this paper, the authors show how PER can be used to model dependencies in programs. Indeed, as we noted above, the problem of non-interference can be seen as absence of dependencies among data, where the meaning of dependency is given by Cohen [19]. The idea behind this characterization consists in interpreting security types as partial equivalence relations. In particular the variables $\mathtt{H}$ on D are interpreted by using the equivalence relation All_D, and $\mathtt{L}$ by using the relation Id_D, where for all $x, x' \in D$:

$$x\ All_D\ x' \qquad\qquad x\ Id_D\ x' \ \Leftrightarrow\ x = x'$$

The intuition behind the relations All_D and Id_D is that they represent the perspective of the user who does not have access to the high information. This user can see the difference between distinct low data, but any high datum is indistinguishable from any other. This perspective can simply be generalized to multilevel security problems.

In order to use this model in the security framework, consider partial equivalence relation, namely equivalence relation which can fail the reflexive property. At this point, we can define a relation between functions. Let $Per(D)$ be the set of partial equivalence relations on D. Given $\mathtt{P} \in Per(D)$ and $\mathtt{Q} \in Per(E)$ we can define $(\mathtt{P} \twoheadrightarrow \mathtt{Q}) \in Per(D \longrightarrow E)$:

$$f\,(\mathtt{P} \twoheadrightarrow \mathtt{Q})g \;\Leftrightarrow\; \forall x, x' \in D \,.\, x \,\mathtt{P}\, x' \;\Rightarrow\; f(x)\,\mathtt{Q}\; g(x')$$

which is in general partial since it can fail reflexivity. Consider $\mathtt{P} \in Per(D)$, if $x \in D$ is such that $x\,\mathtt{P}\,x$ then we write $x : \mathtt{P}$. Therefore, if f is such that $\forall x, x' \in D \,.\, x \,\mathtt{P}\, x' \;\Rightarrow\; f(x)\,\mathtt{Q}\; f(x')$ then we write $f : \mathtt{P} \twoheadrightarrow \mathtt{Q}$. Finally for binary relations $\mathtt{P}$ and $\mathtt{Q}$, we define the relation $\mathtt{P} \times \mathtt{Q}$ by:

$$\langle x, y\rangle\; \mathtt{P} \times \mathtt{Q}\; \langle x', y'\rangle \;\Leftrightarrow x \,\mathtt{P}\, x' \;\wedge\; y \,\mathtt{Q}\, y'$$

At this point, we can formalize security in this model: let us distinguish, in the state s of P the values for low and private variables, i.e., $s = \langle s^{\mathtt{H}}, s^{\mathtt{L}}\rangle$ and let P be a program and $[\![P]\!]$ its semantics, then P is *secure* iff

$$[\![P]\!] : All \times Id \twoheadrightarrow All \times Id \;\equiv\; \forall s, t. \langle s^{\mathtt{H}}, s^{\mathtt{L}}\rangle All \times Id \langle t^{\mathtt{H}}, t^{\mathtt{L}}\rangle \;\Rightarrow\; [\![P]\!](s) All \times Id [\![P]\!](t)$$

where clearly $[\![P]\!](s)$ returns a state which is again a tuple of low and private values.

5.2 Background: Enforcing non-interference

Belief that a system is secure, with respect to confidentiality, should arise from a rigorous analysis showing that the system, as a whole, *enforces* the confidentiality policies of its users. In particular, we are interested in enforcing information flow policies. With the term *enforcement* we mean the checking process that ensures that a program does not reveal private information [80]. There are several approaches for checking non-interference. The standard method used for checking non-interference is to show that an attacker cannot observe any difference between two executions that differ only in the confidential input [69]. Clearly information flow analysis methods can be used for this purpose, but other approaches can be studied and developed. Statically, we can enforce non interference by using a type system. The idea is that of augmenting the type of variables and expressions with annotations that specify policies on the use of typed data, in order to enforce security policies at compile time. Other approaches define a semantic-based security model, providing powerful reasoning techniques. Checking non-interference is indeed an abstraction of the rigorous notion of non-interference that we want to enforce. In particular multi-level security can be expressed at three levels of abstraction [69]:

1. As a precise security policy, defined by a simple security requirement on languages, like the one given above;
2. As a set of general conditions on the transition function of the system that inductively guarantees its multi-level security, as in Bell-LaPadula model [13];
3. As a finite set of lemmas obtained by syntactic analysis of system specifications, such that if all the lemmas are true then any system satisfying these specifications is guaranteed multi-level secure with complete mathematical certainty.

The first formulation is the closest to intuition, since it expresses directly the constraints that should be enforced on the information flow, i.e., it expresses the policy that has to be enforced. The second formulation, which is obtained as *unwinding* [69] of the first one, reduces the proof of satisfaction of the policy to simpler conditions that, by inductive argument, guarantee that the policy holds. Finally, the third formulation is such that, if the process of derivation of lemmas from the policy has been proved mathematically sound, then it reduces the problem of obtaining full mathematical certainty about the security of a system to a form that can be checked by a theorem-prover.

5.2.1 Standard security mechanism

As we noted in the introduction, the standard mechanism used for checking non-interference is *access control.* Access control, which consists in a collection of access control lists and capabilities, is an important part of language-based security. For instance, it can be used when a file may be assigned access control permissions that prevent users, other than its owner, from reading the file. One of the most famous models, based on access control, is the Bell and LaPadula model [13] (see below).
The problem with access control is that it cannot control how data are propagated after they have been read from the file. For this reason, access control is not sufficient for guaranteeing certain kind of security policies, and information-flow control has to be used. Other common mechanisms are, for example, firewalls, encryption and antivirus software which can be used for protecting confidential information. The problem with these mechanisms is that they do not provide end-to-end security. For example, with encryption, we have not assurance that, after decryption, the confidentiality of data is respected. Another problem, related to access control mechanisms, is that it has been proved undecidable whether an access right to an object will "leak" to a process in a system whose access control mechanism is modeled by an access matrix [73].

Bell and LaPadula model.

Bell and LaPadula use finite-state machines to formalize their model [13]. They define the various components of the finite state machine defining what it means (formally) for a given state to be secure. In particular, they consider only the transitions that can be allowed so that a secure state can never lead to an insecure one. This model is based on the access matrix model which is composed by an *access matrix* that decides in which mode each *subject* (user, program,...) can access to an *object* (files, variables,...). In addition to subjects and objects of the access matrix, the Bell and LaPadula model includes the security levels of the system: each subject has a *clearance* and each object has a *classification*. Each object has also a current security level which has not to exceed the subject's clearance. At this point, a set of rules, governing the transitions among states, is used in order to preserve the given security properties. Each rule is formally defined and it is provided together with a set of restrictions on the possible applications of the rule itself.

5.2.2 Denning and Denning Information flow static analysis

One of the first work which aim is to provide a mathematical framework suitable for formulating the requirement of secure information flow is [38]. The central component of this model is a lattice structure derived from the security classes and justified by the semantics of information flow. Security here means that *no unauthorized flow of information is possible*, which is another formulation for non-interference.

Consider an information flow model $\mathcal{F}$, defined as $\mathcal{F} = \langle N, P, \mathcal{S}, \oplus, \rightarrow \rangle$, where $N = \{a, b, \ldots\}$ is a set of objects, $P = \{p, q, \ldots\}$ is a set of processes, which are active agents responsible of information flow. $\mathcal{S} = \{A, B, \ldots\}$ is a complete lattice of *security classes* corresponding to disjoint classes of information, with least upper bound $\oplus$ and greatest lower bound denoted by $\otimes$. The idea behind these classes is that of modeling the security classification of objects. Each object a is bounded to a security class A which specifies the security class associated with the information stored in a. There are two kinds of binding: *static*, where the security classes associated with objects are constants, and *dynamic*, where the security classes may vary during the execution. The operator $\oplus$ is the class-combining operator, an associative and commutative binary operation that specifies, given two operand classes, the class in which the result of any binary function on values from the operand classes belongs. Finally $\rightarrow$ is a flow relation among classes. Given two classes A and B, we write $A \rightarrow B$ if information in class A is permitted to flow into class B. *Information is said to flow from class A to class B whenever information associated with A affects the value of information associated with B.* At this point, a flow model $\mathcal{F}$ is secure if and only if the execution of a sequence of operations cannot give rise to a flow that violates the relation $\rightarrow$.

Enforcing security.

The primary problem in guaranteeing security lies in detecting (and monitoring) all flows causing a variation of data [38]. Here, we find the first distinction between *implicit* and *explicit* flows. Consider the statement **if** $a = 0$ **then** $b := 0$ **else nil**; if initially $b \neq 0$ then we can know something about a after the execution of the statement. For this reason the authors distinguish between *implicit* and *explicit* flows. Explicit flows are those due to the execution of any statement that directly transfer information among variables. Implicit flows to b occur when the result of executing or not a statement, that causes an explicit flow to b, is conditioned on the value of a guard. At this point, in order to specify the security requirements of programs causing implicit flows, it is convenient to consider an abstract representation of programs that preserves the flows but not necessarily the whole original structure. The abstract program S is defined recursively:

1. S is an elementary statement, i.e., an assignment;
2. There exist S_1 and S_2 such that $S = S_1; S_2$;
3. There exist $S_1, \ldots, S_m$ and an m-valued variable c such that $S = c : S_1, \ldots, S_m$.

where the third point defines conditional structures in which the value of a variable selects among alternative programs.
At this point, security is enforced by modeling implicit and explicit flows in the lattice of security classes and checking if these flows are allowed by the security policy chosen. Let us see how this is defined for the abstract program S described above. An elementary statement S is secure if any explicit flow caused by S is secure, namely if the value of b is derived in S from the values of $a_1, \ldots, a_m$ then $A_1 \oplus \ldots \oplus A_m \to B$ is allowed. $S = S_1; S_2$ is secure if both S_1 and S_2 are secure. Finally $S = c : S_1, \ldots, S_m$ is secure if each S_k is secure and all implicit flows from c are secure, namely let $b_1, \ldots, b_m$ be the objects into which S specifies explicit flows, then all the implicit flows are from c to each b_k and they are secure if $C \to B_1 \otimes \ldots \otimes B_m$ is allowed.
The authors use this model for generating a certification mechanism for secure information flow [39]. In particular, in the hypothesis of static binding, they easily incorporate the certification process into the analysis phase of a compiler and the mechanism is presented in the form of certification semantics - actions for the compiler to perform, together with usual semantic actions such as type checking and code generation, when a string of a given semantic type is recognized. This analysis has been widely studied and has been characterized as an extension of an axiomatic logic for program correctness in [7] (see Sect. 5.2.4).

5.2.3 Security type systems

A *security type system* is a collection of inference rules and axioms for deriving typing judgments, in particular it describes which security type is assigned to a program (or expression), based on the types of subprograms (or subexpressions). In [114] the Denning's approach is formulated as a type system, in such a way that all the well-typed programs are proved satisfy the non-interference property. A typing judgment has the form:

$$\gamma \vdash n : \tau \qquad \gamma \vdash x : \tau\ var \qquad \frac{\gamma \vdash e : \tau\ var}{\gamma \vdash e : \tau} \qquad \frac{\gamma \vdash x : \tau\ var \quad \gamma \vdash e : \tau}{\gamma \vdash x := e : \tau\ cmd}$$

$$\frac{\gamma \vdash c_1 : \tau\ cmd \quad \gamma \vdash c_2 : \tau\ cmd}{\gamma \vdash c_1; c_2 : \tau\ cmd} \qquad \frac{\gamma \vdash e : \tau \quad \gamma \vdash c : \tau\ cmd \quad \gamma \vdash c' : \tau\ cmd}{\gamma \vdash \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c' : \tau\ cmd}$$

$$\frac{\gamma \vdash e : \tau \quad \gamma \vdash c : \tau\ cmd}{\gamma \vdash \textbf{while}\ e\ \textbf{do}\ c}$$

Table 5.1. Security type system

$$\frac{\tau \leq \tau'}{\vdash \tau \subseteq \tau'} \qquad \vdash \rho \subseteq \rho \qquad \frac{\vdash \rho \subseteq \rho', \ \vdash \rho' \subseteq \rho''}{\vdash \rho \subseteq \rho''}$$

$$\frac{\vdash \tau \subseteq \tau'}{\vdash \tau\ cmd \supseteq \tau'\ cmd} \qquad \frac{\gamma \vdash p : \rho, \ \vdash \rho \subseteq \rho'}{\gamma \vdash p : \rho'}$$

Table 5.2. Subtyping rules

$$\gamma \vdash p : \tau$$

which asserts that the program p has type τ with respect to the identifier typing γ. An identifier typing is a map from identifiers to types; it gives the type of any free identifier of p. So, for example, we have the inference rule $\gamma \vdash x : \tau$ if $\gamma(x) = \tau$. Let's start from the Denning's model [38]. The types of a systems are stratified into two levels. At one level are *data types*, denoted by τ, which are the security classes of $\mathcal{S}$, partially ordered by $\leq$. At the other level are *phrase types*, denoted by ρ. These include data types, assigned to expression, variable types of the form $\tau\ var$ and command types of the form $\tau\ cmd$. As expected, a variable type $\tau\ var$ stores information whose security class is τ or lower. Moreover, a command c has type $\tau\ cmd$ only if it is guaranteed that every assignment within c is made to a variable whose security class is $\tau\ var$ or higher. This is the confinement property ensuring secure implicit flows. In order to formalize this relation we have to extend the partial order $\leq$ on security classes to a subtype relation $\subseteq$ among types. A simplified version of the rules introduced in [114] are given in Table 5.1. In Table 5.2 we can find the subtyping rules. This system has been proved to be sound and therefore each program that can be typed in this system has only secure information flow. On the other hand, the system is not complete, which means that there are programs with only secure information flows and that cannot be typed in this system. For instance, the program $p \overset{\text{def}}{=} \textbf{if}\ h = 1\ \textbf{then}\ l := 0\ \textbf{else}\ l := 0$ with $l : \mathtt{L}$, $h : \mathtt{H}$ and $\mathtt{L} \leq \mathtt{H}$, has clearly only secure information flows but it cannot be typed in the system in Table 5.1.

5.2.4 The axiomatic approach

Another important approach for checking the existence of insecure information flows is the axiomatic one introduced, for the first time, in [7]. This approach uses a program *flow proof* constructed applying flow axioms and inference rules. An important aspect of this technique is that it can certify flows in both parallel and sequential programs. Moreover, once the flow proof for a program has been constructed, the proof can be used to validate a variety of flow policies. The idea of this work consists in using assertions of the kind $\{P\}\,S\,\{Q\}$, which means that if P is true before the execution of S, then Q is true after the execution of S, provided that S terminates. This is the standard notation used in correctness proofs, the difference is that P and Q here refers to classes rather than to values. In order to develop a *flow proof* of $\{P\}\ S\ \{Q\}$, the authors describe a deductive logic that allows to characterize the information flow semantics of statements. The inference rules used are of the form

$$\frac{A_1, \ldots, A_n}{B}$$

which means that if logical statements A_i are true, then so is B.

More recently, in [6], another Hoare-style logic has been defined in order to analyze information flow for confidentiality. In this case, confidentiality is treated as *independency* of variables [19], and program traces, potentially infinitely many, are abstracted, in the standard framework of abstract interpretation [28], by a finite set of variable independencies. The potentiality of this approach is that these variable independencies can be statically checked against the logic. Moreover, this method allows, once a program is deemed insecure, to explain why the program is insecure by statically generating counterexamples. The basic idea of this paper is to annotate the program in order to statically check independencies. This is achieved by using the Hoare-like logic described in Table 5.3, where $[x\#w]$ denotes that the current value of x is independent of the initial value of w, and where judgements are of the form $G \vdash \{T_1^{\#}\}\ C\ \{T_2^{\#}\}$. This judgement is interpreted by saying that if the independencies described in $T_1^{\#}$ hold *before* execution of C, then the independencies described in $T_2^{\#}$ will hold *after* the execution of C, provided that C terminates. In [6], the authors provide also a correctness result, which can be seen as the non-interference result for information flow. Indeed, with l and h interpreted as low and high respectively, suppose that $[l\#h]$ appears in the final set of independencies $T^{\#}$, after the execution of a program C. Then, any two traces in the execution of C, that have initial values that differ only on h, must agree on the current value of l. Moreover, if, on the other hand, the program is deemed insecure, i.e, $[l\#h]$ does not appear in the final set of independencies, then it means that l is dependant on h, and, in addition, the derived assertions allow to find a counterexample, i.e., two initial values of h that produce two different final values of l.

[Assign] $G \vdash \{T_0^{\#}\}x := e\{T^{\#}\}$

If $\forall[y\#w] \in T^{\#}$.
$(x \neq y \;\Rightarrow\; [y\#w] \in T_0^{\#})$,
$(x = y \;\Rightarrow\; w \notin G \;\wedge\; \forall z$ free variable in $e.\,.\,[z\#w \in T_0^{\#}])$

[Seq]
$$\frac{G \vdash \{T_0^{\#}\}C_1\{T_1^{\#}\},\ G \vdash \{T_1^{\#}\}C_2\{T_2^{\#}\}}{G \vdash \{T_0^{\#}\}C_1; C_2\{T_2^{\#}\}}$$

[If]
$$\frac{G_0 \vdash \{T_0^{\#}\}C_1\{T^{\#}\},\ G_0 \vdash \{T_0^{\#}\}C_2\{T^{\#}\}}{G \vdash \{T_0^{\#}\}\textbf{if } e \textbf{ then } C_1 \textbf{ else } C_2\{T^{\#}\}}$$

If $G \subseteq G_0$,
$w \notin G_0 \;\Rightarrow\; \forall x$ free variable in $e\,.\,[x\#w] \in T_0^{\#}$

[While]
$$\frac{G_0 \vdash \{T^{\#}\}C\{T^{\#}\}}{G \vdash \{T^{\#}\}\textbf{while } e \textbf{ do } C\{T^{\#}\}}$$

If $G \subseteq G_0$,
$w \notin G_0 \;\Rightarrow\; \forall x$ free variable in $e\,.\,[x\#w] \in T^{\#}$

[Sub]
$$\frac{G_1 \vdash \{T_1^{\#}\}C\{T_2^{\#}\}}{G_0 \vdash \{T_0^{\#}\}C\{T_3^{\#}\}}$$

If $T_0^{\#} \subseteq T_1^{\#},\ T_2^{\#} \subseteq T_3^{\#},\ G_0 \subseteq G_1$

Table 5.3. An axiomatic logic for independencies

5.3 Non-interference for different computational systems

A major line of research in information flow purses the goal of defining non interference for the different computational models, and for accommodating the increased expressiveness of modern programming languages.

5.3.1 Deterministic systems: Imperative languages

As we underlined before, non-interference for *programs* essentially means that any possible variation of confidential (high/private) input does not cause a variation of public (low) output. This in particular means that each variable has a static attribute called *security level.* In [114] the confinement property for deterministic languages is defined as follows.

Definition 5.1. *A program P has the* non-interference *property if for all memories μ and ν such that $\mu(l) = \nu(l)$ for all low variables l, and such that P terminates* *successfully starting both from μ and ν, yielding, respectively, to μ' and to ν', then we have $\mu'(l) = \nu'(l)$ for all low variables l.*

Basically, it says that altering the initial contents of private variables does not interfere with the final value of any low variable. For instance, if variable PIN is private and y is public then the following program does not preserve confinement, exactly as the program $y := \text{PIN}$:

```
while $\neg(mask = 0)$
    if $\neg(\text{PIN} \,\&\, mask = 0)$ (bitwise and)
        $y := y \mid mask$;    (bitwise or)
    $mask := mask/2$;
```

If $mask$ is a power of two, then it indirectly copies PIN to y, one bit at time [114].

Starting from the Cohen's seminal study of strong dependency [19], the notion of non-interference can be rigorously formalized using the programming-language semantics. Suppose that $s \in \Sigma$ is the denotation for states of programs, and that states, representing the tuples of values assigned to variables (i.e., representing memories), can be partitioned in order to distinguish the values of private variables from the values of public ones: $s = \langle s^{\mathrm{H}}, s^{\mathrm{L}} \rangle$. In general a program, starting from a state s can terminate in a state s' or can diverge. The denotational semantics of programs is the function that associates with each possible initial state the set of all the corresponding terminal state together with $\perp$, if the given initial state can lead to non-termination. Moreover, we can define an equivalence relation among states: $s_1 =_{\mathrm{L}} s_2$ iff $s_1^{\mathrm{L}} = s_2^{\mathrm{L}}$. Therefore, for a given semantic model $[\![P]\!]$ of the program P, non-interference can be formalized as follows: P is secure iff

$$\forall s_1, s_2 \in \Sigma \,.\, s_1 =_{\mathrm{L}} s_2 \;\Rightarrow\; [\![P]\!](s_1) =_{\mathrm{L}} [\![P]\!](s_2) \tag{5.2}$$

which is exactly the absence of strong dependency of public data from private ones [19]. For example the program

$$c \overset{\text{def}}{=} \textbf{if } h = 3 \textbf{ then } l := 5 \textbf{ else nil}$$

is clearly insecure since the high initial values 3 and 4 provides different results for the variable l: $\langle 4, 1\rangle =_{\mathrm{L}} \langle 3, 1\rangle$ but $[\![c]\!](4,1) = \langle 4, 1\rangle$ while $[\![c]\!](3,1) = \langle 3, 5\rangle$, where $\langle 4, 1\rangle \neq_{\mathrm{L}} \langle 3, 5\rangle$.
In general we can rewrite non-interference by saying that if two state share the same low values, then the behaviours of the program executed on these states are indistinguishable by the attacker. This means that the notion can be made parametric on what the attacker can really see. This is a key observation in order to abstract the notion of non-interference.

5.3.2 Non-deterministic and thread-concurrent systems

The natural extension of the notion of non-interference to non-deterministic systems is the notion of *possibilistic* non-interference [86]. As we have said before, in order to prevent direct information flows, certain aspects of the system behaviour must not be directly observable by users who do not have the appropriate clearance. However, in general, an observer might still be able to deduce confidential information from other observations. In the worst case, the observer has complete knowledge of the system and can construct all the possible system behaviours which generate a given observation, trying to deduce confidential information from this set. The basic idea of possibilistic security is to demand that this set is so large that the observer cannot deduce confidential information since it cannot be sure which behaviour has actually occurred [86]. In [108] the confinement (non-interference) property for non-deterministic languages is defined as:

Definition 5.2. *A non-deterministic program P satisfies the* possibilistic non-interference *property if for all memories μ and ν such that $\mu(l) = \nu(l)$ for all low variables l, and P can terminate successfully starting from μ yielding to the final state μ', then there exists a state ν' such that P can terminate successfully starting from ν yielding ν' and $\mu(l) = \nu'(l)$ for all low variables l.*

It says that altering the initial contents of high variables does not interfere with the *set of possible final values* of any low variable [108]. The property rules out concurrent programs with information channels that exploit thread synchronization. In particular, we have a purely non-deterministic system if the scheduler of the system, that activates the threads, is characterized by the simple rule: *At each step, any thread can be selected to run for one step.* For instance, consider the following system:

Thread α:	Thread β:	Thread γ:
$y := x;$	$y := 0;$	$y := 1$

Suppose that x is a private binary variable, while y is public. Then the program satisfies the possibilistic non-interference property.

Possibilistic security properties.

Due the complex structure of non-deterministic systems, the notion of possibilistic non-interference given above, is not the only *confidentiality property* that can be defined on this kind of systems. The first attempts to provide a general theory in which uniformly define possibilistic security properties was through the use of *selective interleaving functions* [91]. In this paper, it is observed that possibilistic security properties fall outside of the Alpern-Schneider safety/liveness domain [4], since these properties are not properties of traces, i.e., trace sets, but properties of trace sets, i.e., sets of trace sets. In particular, possibilistic security properties are defined as *closure properties with respect to some functions that takes two traces and interleaves them to form a third trace* [91]. This theory is then used for studying how these security properties behave when systems are composed, i.e., if a system satisfying property X is composed with a system satisfying property

Y, using composition constructor Z, what properties will the composite system satisfy? In the following we will recall the principal security properties treated in this general theory.

Non-inference: Informally, non-inference requires that for any trace of the system, removing all the high level events, we obtain a trace that is still valid. More formally, if *purge*(τ) is the function that takes a trace τ and sets all high level inputs and outputs in τ to the empty value λ, then a system satisfies non-inference if the set of its traces is closed under the function *purge*, i.e., the image of *purge* is always contained in the set of valid traces of computation.

Generalized Non-inference: Informally, generalized non-inference requires that for any trace τ, it must be possible to find another trace σ such that the low level events of τ are equal to σ and σ has not high level inputs. More formally, if *input-purge*(τ) is the function that takes a trace τ and sets all high level inputs in τ to the empty value λ, then a system satisfies non-inference if the set of its traces is closed under the function *input-purge*.

Separability: Informally, separability holds if no interaction is allowed between high level and low level events. It is like having two separate systems, one running the high level processes, and one running the low level ones. More formally, if *interleave*(τ_1, τ_2) is the function that takes two traces τ_1 and τ_2 and returns the trace τ such that the high input and output of τ are taken in τ_1 and low input and output of τ are taken in τ_2, then a system satisfies separability if the set of its traces is closed under the function *interleave*.

Generalized Non-interference: Generalized non-interference holds if modifying a trace τ, inserting or deleting high level input, results in a sequence σ that can be transformed in a valid trace by inserting or deleting high level outputs. More formally, if *input-interleave*(τ_1, τ_2) is the function that takes two traces τ_1 and τ_2 and returns the trace τ such that the high input of τ are taken in τ_1 and low input and output of τ are taken in τ_2, then a system satisfies generalized non-interference if the set of its traces is closed under the function *input-interleave*.

This framework has been made more intuitive and general in [117], in order to model more security properties, such as *perfect security property* (PSP), which allows high level outputs to be influenced by low level events [117]. More recently, all these security properties have been modeled in a *modular structure* in [85], where they are obtained as combination of *basic security predicates*.

5.3.3 Communicating systems: Process algebras

The possibilistic notions of non-interference introduced in the previous section allows to consider non deterministic system, but are not adequate for treating non-interference in systems with the *synchrony* assumption: a system is composed of several components which have to proceed together at every time instant [47].

Synchrony is a basic feature, together with non-determinism, of *process algebras*, and probably, the most famous representative of this class is CCS [92]. In particular, in [47], the problem of studying secure information flows is considered in a particular process algebra, SPA (see Sect. 4.2.2), which is a slight extension of CCS. At this point, we recall the principal notions of non-interference defined on SPA in [47]. In particular there are two classes of definitions, depending on the equivalence of processes chosen: trace-based or bisimulation-based. In order to better understand the notions of non-interference that we are going to introduce, let's reformulate the idea of non-interference as follows: *Let G and G' be two user groups, given any input sequence γ, let γ' be its subsequence obtained by deleting all the actions of users in G; G is non-interfering with G' iff for every input sequence γ, the users of G' obtain the same output after the execution of γ and of γ'.*

Trace-based security properties.

Let us consider the trace-based equivalence of processes $\approx_T$, i.e., $A_1 \approx_T A_2$ iff the set of traces associated with A_1 is equal to the set of traces associated with A_2. Then the first extension of the notion of non-interference to SPA is the *Non-deterministic Non-Interference* (NNI), defined as follows:

$$A \in NNI \quad \Leftrightarrow \quad (A\backslash_I Act_{\mathtt{H}})/Act_{\mathtt{H}} \approx_T A/Act_{\mathtt{H}}$$

This notion requires that, when we avoid high level inputs, we obtain a trace whose projection on low level actions (i.e., the hiding of high level actions) is equal to the low level projection of a generic trace of actions of the system. A more restrictive form of NNI requires that, for every trace γ, the sequence γ', obtained deleting all the high level actions (input and output), is still a trace. This property is called *Strong NNI* (SNNI) and is defined as follows:

$$A \in SNNI \quad \Leftrightarrow \quad A/Act_{\mathtt{H}} \approx_T A\backslash Act_{\mathtt{H}}$$

The relation between these two notions is that, in SPA, $SNNI \subset NNI$. If, such as in CSP, we don't have distinction between inputs and outputs, then $NNI = SNNI$.
Another interesting notion of non-interference is *Non-Deducibility on Compositions* (NDC). A system is NDC if the set of its low level views cannot be modified by composing the system with any high level process. This property can be defined as follows:

$$A \in NDC \quad \Leftrightarrow \quad \forall \Pi \in \mathcal{E}_{\mathtt{H}} \ . \ A/Act_{\mathtt{H}} \approx_T (A \parallel \Pi)\backslash Act_{\mathtt{H}}$$

In [47] it is proved that $NDC = SNNI$.

Bisimulation-based security properties.

All the security notions introduced so far are based on the assumption that the semantics of a system is the set of its execution traces. In this section we show that, in [47], these security properties have been rephrased on the finer notion of system behaviour called *weak bisimulation* (or observational equivalence) [92]. This extension was considered since trace semantics is rather weak, as it is unable to distinguish systems which give different observations to a user, even if they have the same traces. Here we recall the definition of weak bisimulation over SPA agents [47]. Let $A \overset{\mu}{\Longrightarrow} A'$ a short hand for $A \xrightarrow{\tau}{}^* A_1 \xrightarrow{\mu} A_2 \xrightarrow{\tau}{}^* A'$, where $\xrightarrow{\tau}{}^*$ means zero or more times τ. In the following $A \overset{\hat{\mu}}{\Longrightarrow} E'$ stands for $A \overset{\mu}{\Longrightarrow} A'$ if $\mu \in \mathcal{L}$, for $A \xrightarrow{\tau}{}^* A'$ if $\mu = \tau$. The following example shows that trace-based equivalence is weaker than bisimulation based equivalence. Indeed the two systems have the same set of traces but they are not bisimilar.

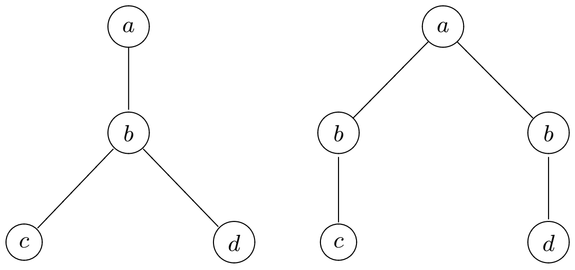

Fig. 5.1. Trace vs bisimulation equivalence

Definition 5.3. *A relation $R \subseteq \mathcal{E} \times \mathcal{E}$ is a* weak bisimulation *if it satisfies:*

- *Whenever $\langle A, B\rangle \in R$ and $A \xrightarrow{\mu} A'$, then there exists $B' \in \mathcal{E}$ such that $B \overset{\hat{\mu}}{\Longrightarrow} B'$, and $\langle A', B'\rangle \in R$;*
- *Whenever $\langle A, B\rangle \in R$ and $B \xrightarrow{\mu} B'$, then there exists $A' \in \mathcal{E}$ such that $A \overset{\hat{\mu}}{\Longrightarrow} A'$, and $\langle A', B'\rangle \in R$;*

Two SPA *agents $A, B \in \mathcal{E}$ are observationally equivalent, $A \approx_B B$, if there exists a weak bisimulation containing the pair $\langle A, B\rangle$.*

Note that $\approx_B$ is an equivalence relation, and that it is stronger than $\approx_T$. At this point in [47] the *Bisimulation NNI* (BNNI), *Bisimulation SNNI* (BSNNI) and the *Bisimulation NDC* (BNDC) are introduced simply by substituting $\approx_B$ for $\approx_T$ in their algebraic SPA-based characterizations. In particular we can give a characterization of BNDC equivalent to the simple substitution of the equivalence relation.

- $A \in BNNI$ iff $(A \backslash_I Act_{\mathtt{H}})/Act_{\mathtt{H}} \approx_B A/Act_{\mathtt{H}}$;
- $A \in BSNNI$ iff $A/Act_{\mathtt{H}} \approx_B A\backslash Act_{\mathtt{H}}$;
- $A \in BNDC$ iff $\forall \Pi \in \mathcal{E}_{\mathtt{H}} \,.\, A\backslash Act_{\mathtt{H}} \approx_B (A \parallel \Pi)\backslash Act_{\mathtt{H}}$.

All the relations among these notions are deeply studied in [47].

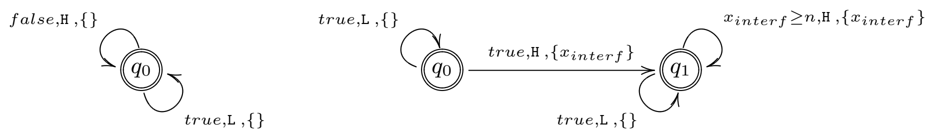

Fig. 5.2. The automata $Inhib_{\mathtt{H}}$ and $Interf^{n}_{\mathtt{H}}$.

5.3.4 Real-time systems: Timed automata

The most widespread models for real-time systems are timed automata (see Sect. 4.2.3). In [12], a new notion of non-interference for timed automata is introduced. The notion is based on high-level action delays magnitude and on equivalence of timed automata. Given a natural number n, the authors say that *high-level actions do not interfere with the system, considering minimum delay n, if the system behaviour in absence of high-level actions is equivalent to the system behaviour, observed on low-level actions, when high-level actions can occur with a delay between them greater than or equal to n.* Thus, the environment of the system does not offer high-level events separated by less that n times units, and if the property holds, there is no way for low-level users to detect any high-level action. The main improvement of this notion, if compared with untimed notions, is that time is observable and the property captures those systems in which the time delay between high-level actions cannot be used to construct illegal information flows.
Let A be a timed automaton over the alphabet of actions Σ and $\langle\!| A |\!\rangle$ the accepted language associated with A. We suppose that Σ is partitioned into two disjoint sets of actions `H` and `L` such that `H` is the set of the high-level actions, while `L` is the set of the low-level ones. First of all, consider an automaton A, we want to observe its behaviour in absence of high-level actions. In order to obtain this, we compose it in parallel with an automaton, called $Inhib_{\mathtt{H}}$, that does not allow the execution of high-level actions (see Fig. 5.2). In Fig. 5.2 we use the conventions that double-circled states are final, and q_0 is initial, moreover, an edge having as label a set of actions represents a set of edges, one for each action in the set, with the same clock constraint and clock reset. In the product $A||Inhib_{\mathtt{H}}$ the component A cannot have transition labeled by $h \in \mathtt{H}$ since $Inhib_{\mathtt{H}}$ never performs high-level actions (its constraints on high-level actions are false). Thus only low-level actions are executed.
Consider $Interf^{n}_{\mathtt{H}}$ in Fig. 5.2. This automaton allows the execution of high-level actions only when they are separated by at least n time units. Indeed, both the states can execute low-level actions without any restriction. But, if a high-level action occurs, then the automaton goes in state q_1 and reset the clock x_{interf}, which is reset by all high-level actions, and all high-level actions can be executed if x_{interf} is greater or equal than n. Namely a high-level action can be executed only if at least n time units have elapsed from the previous one.

Then an automaton A is said to be n *non-interfering* if:

$$(A||Interf^n_{\mathtt{H}})/\mathtt{H} \approx A||Inhib_{\mathtt{H}}$$

where the operator $/\mathtt{H}$ hides high-level actions, namely whenever the label of an edge is $\sigma \in \mathtt{H}$ it is replaced by ε, and $\approx$ is defined by: $A_1 \approx A_2$ iff $\mathcal{L}(A_1) = \mathcal{L}(A_2)$.

The notion of non-interference for timed automata can be equivalently characterized on languages [12]. Let $\langle\!| A |\!\rangle$ be the timed language accepted by A on a alphabet Σ and consider the following manipulation of languages:

$$\begin{aligned}
\langle\!| A |\!\rangle|_{\mathtt{L}} &\stackrel{\text{def}}{=} \left\{ \overline{\langle\sigma, t\rangle} \in \langle\!| A |\!\rangle \,\middle|\, \forall\langle\sigma_i, t_i\rangle \in \overline{\langle\sigma, t\rangle} \,.\, \sigma_i \in \mathtt{L} \right\} \\
\langle\!| A |\!\rangle/\mathtt{H} &\stackrel{\text{def}}{=} \left\{ \omega \,\middle|\, \begin{array}{l} \exists\overline{\langle\sigma, t\rangle} \in \langle\!| A |\!\rangle \text{ such that } \omega \text{ is the projection of } \overline{\langle\sigma, t\rangle} \\ \text{on the pairs } \left\{ \langle\sigma, t\rangle \,\middle|\, \sigma \in \mathtt{L} \right\} \end{array} \right\} \\
\langle\!| A |\!\rangle^n_{\mathtt{H}} &\stackrel{\text{def}}{=} \left\{ \overline{\langle\sigma, t\rangle} \in \langle\!| A |\!\rangle \,\middle|\, \begin{array}{l} \forall\langle\sigma_i, t_i\rangle, \langle\sigma_j, t_j\rangle \in \overline{\langle\sigma, t\rangle} \,.\, i \neq j,\ \sigma_i, \sigma_j \in \mathtt{H} \\ \Rightarrow\ |t_i - t_j| \geq n \end{array} \right\}
\end{aligned}$$

Namely, $\langle\!| A |\!\rangle|_{\mathtt{L}}$ avoids high-level actions, it takes only the traces of the system that make only low-level actions. On the other hand, $\langle\!| A |\!\rangle/\mathtt{H}$ hides the high-level actions, i.e., it executes them and then it hides them. Finally, $\langle\!| A |\!\rangle^n_{\mathtt{H}}$ selects only those traces where the high-level actions are distant at least n.
Then, in [12], a system is said to be *n-non-interfering* iff

$$\langle\!| A |\!\rangle^n_{\mathtt{H}}/\mathtt{H} = \langle\!| A |\!\rangle|_{\mathtt{L}}$$

5.4 Covert Channels

By covert channels we mean those channels that are not intended for information transfer at all [80]. The importance of studying these kind of channels lies on the fact that they pose the greatest challenge in preventing improper transmission leaks. There are several kind of covert channels [104]:

Implicit channels : Channels of information flow due to the control structure of a program;

Termination channels : Channels of information flow due to the termination or non-termination status of a program;

Timing channels : Channels of information flow due to the time at which an action occurs rather than due to the data associated with the action. The action may be termination;

Probabilistic channels : Channels of information flow due to the change of the probability distribution of observable data. These channels are dangerous when the attacker can run repeatedly a computation and observe its stochastic behaviour;

Resources channels : Channels of information flow due to the possible exhaustion of a finite, shared resource, such as disk memory;

The kind of covert channel, that may be created, depends on what the attacker/user can view of the computational system. This means that a computational system can be said to protect confidential information only with respect to a model of what attackers/users are able to observe of its execution.

5.4.1 Termination channels

Consider Definition 5.1 of non-interference for deterministic languages. In this definition it is said that the program has to "terminate successfully", starting from the given states. It is clear that, changes in high variables may cause the program to diverge, leaving unchanged the fact that the program can still satisfy the definition. This may make the property unsuitable in situations where this sort of behaviour can be observed. If, such as for PER model, the denotational semantics is used for defining non-interference, then we note that in case of non-termination denotational semantics associates with each state, leading to non-termination, the symbol $\perp$. In this way, Eq. 5.2 can be used also for defining termination-sensitive non-interference. Therefore, the PERs model can be simply adapted by considering domains of values enriched with the symbol $\perp$, i.e., $D_\perp$, and extending relations $\mathtt{R} \in \mathit{Per}(D)$ to $\mathtt{R}_\perp \in \mathit{Per}(D_\perp)$ naturally by adding the relation $\perp \mathtt{R}_\perp \perp$. In this way we make the definition insensitive to non-termination [106]. Namely *termination channels* are avoided simply by enriching the semantics. Note that, also in [1], where for the first time dependencies were given in term of PERs, for a calculus based on a variation of λ calculus, was shown that PERs capture termination sensitive security.
When non-interference is checked on the syntax, by typing secure programs (see Sect. 5.2.3), then it become necessary to enrich the type system in order to avoid termination channels [112]. In this paper, the authors show that termination flows can be handled with just a simple modification of the original type system in [114], based on the notion of *minimum type*. They say that a type τ is minimum if $\tau \leq \tau'$ for every type τ', to handle the covert flow arising from non-termination they merely change the typing rule for **while** b **do** c **endw** to require that b has minimum type. In other words, this means that this type system disallows high loops and require high conditionals have no loops in the branches.

5.4.2 Timing channels

Note that, in practice, non-termination cannot be distinguished from a very time-consuming computation, thus the termination channel can be viewed as an instance of the *timing channel*. Timing-sensitive non-interference can be formalized by considering Eq. 5.2, where the low view relation $=_\mathtt{L}$ is substituted by $\approx_\mathtt{L}$, which relates two behaviour iff both diverges or both terminate in the same number of execution steps in low-equal final states [104]. In [113] the authors avoid timing channels in the type system by restricting high conditionals to have no loops in the branches and wrapping each high conditional in a *protect* statement whose execution is atomic. In [2] *program transformation* is used in order to close timing leaks. In particular, the "type" of a program C is its *low slice* C_{L}, which is syntactically identical to C but only contains assignments to low level variables. All the assignments to high level variables are replaced with appropriate dummy commands with no effect on variables, therefore the low slice has the same observational behaviour as the original program with respect to low level variables. Finally, the usual type system is considered and, either the original program C is rejected (in case of a potential explicit or implicit insecure information flow) or accepted and transformed into the program C_{L} free of timing leaks.

5.4.3 Probabilistic channels

Probability-sensitive non-interference can be formalized in the Eq. 5.2 by replacing $=_{\text{L}}$ with an equivalence relation $\approx_{\text{L}}$ that relates two behaviours iff the distribution of low output is the same. Indeed, as we can see in the following example, possibilistic non-interference is not sufficient to prevent probabilistic information flows [90]. Consider, for example, the following multi-threaded system:

Thread α:	Thread β:
$y := x;$	$y := \mathit{random}(100)$

where $\mathit{random}(100)$ returns a random number between 1 and 100, and $x \in [1, 100]$. Then the program satisfies the possibilistic non-interference since regardless the initial value of x, the final value of y is a random number between 1 and 100. But with a probabilistic semantics, this is not good enough, because the final values of y are not equally probable, indeed the more probable value for y is the initial value of x [109]. Moreover, in multi-threaded systems, also the scheduler of processes may be probabilistic. In [113] the authors define a notion of probabilistic non-interference that captures the probabilistic information flows that may result from a uniform scheduler in a multi-threaded language. In [106] the authors considers PERs on probabilistic powerdomains in order to catch probabilistic flows. While in [105] the authors connect probabilistic security with probabilistic bisimulation [81], improving the precision of the previous probability-sensitive notions.

5.5 Weakening non-interference

The limitation of the notion of non-interference described so far, is that it is an extremely restrictive policy. Indeed, non-interference policies require that *any* change upon confidential data has not to be revealed through the observation of public data. There are at least two problems with this approach. On one side, many real systems are intended to leak some kinds of information. On the other side, even if a system satisfies non-interference, some kinds of tests could reject it as insecure. These observations address the problem of *weakening* the notion of non-interference both characterizing the information that *is allowed* to flow, and considering *weaker* attackers that cannot observe any property of public data. Clearly, as we will show in this thesis, these are dual aspects of the same problem, and in the following sections we will describe the most relevant works in this direction.

5.5.1 Characterizing released information

As we have addressed above, real systems often do leak confidential information, therefore it seems sensible to try to measure that leakage as best as possible. The first work on this direction is [19], where the notion of *selective dependency* (see Sect. 5.1.1) is introduced. Selective dependency consists in a weaker notion of dependency, and therefore of non-interference, that identifies what flows during the execution of programs. More recently, in literature we can find several works that attack this problem from different points of view. A first approach consists in a quantitative (information theoretic) definition of information flows [17, 84]. Another relevant approach models the attacker's power by using equivalence relations, and by transforming these equivalence relations it characterizes the released information [118]. Afterwards, several papers treated the *declassification* of confidential information [83, 96, 103].

An information theory approach.

In [17], Shannon's information theory is used to quantify the amount of information a program may leak and to analyze in which way this depends on the distribution of inputs. In particular, the authors are interested in analysing how much an attacker may learn (about confidential information) by observing the input/output behaviour of a program. The basic idea is that all information in the output of a deterministic program has to come from the input, and what it is not provided by the low input has to be provided by the high input. Therefore, this work wants to investigate how much of the information carried by the high inputs to a program can be learned by observation of the low outputs, assuming that the low inputs are known. Now, since the considered language is deterministic, any variation of the output is due to a variation of the input. Hence, once we account for knowledge of the program's low inputs, the only possible source of *surprise* in an output is the interference from the high inputs. So, given a program variable X (or a set of program variables), let X^ι and X^ω be, respectively, the corresponding random variables on entry and exit from the program. In [17] the authors take as measure of the amount of leakage into X due to the program: $\mathcal{L}(X) = \mathcal{H}(X^\omega | L^\iota)$, where L is the set of low variables, this L^ι is the random variable describing the distribution of the program's non-confidential inputs, and $\mathcal{H}$ is the *entropy*[1]. Moreover, in [17], it is shown that there exists a more general characterization of the amount of information released that is appropriate even for languages with an inherently probabilistic semantics. In this case, they say that a natural definition of the leakage into X is the amount of information shared between the final value of X and the initial value of H, given the initial value of L: $\mathcal{L}' = \mathcal{I}(H^\iota; X^\omega | L^\iota)$, where $\mathcal{I}$ is the *conditional mutual information*[2] between H^ι and X^ω given knowledge of L^ι. This is essentially the definition used by Gray [71], specialized in a simpler semantic setting. In [17] it is also proved that, for deterministic languages $\mathcal{L} = \mathcal{L}'$.

Shannon's information theory is not the only approach, existing in literature, for quantifying information flow. Indeed in [84] the capacity of covert channels, i.e., the *information flow quantity*, is measured in terms of the number of high level behaviours that can be accurately distinguished from the low level point of view. The idea is that if there are N such distinguishable behaviours, then the high level user can use the system to encode an arbitrary number in the range $0, \ldots, N-1$ to send it to the low level user, in other words $\log_2 N$ bits of information are passed.

Declassification.

In the previous paragraph, we described a method that allows to quantify the amount of information released. In literature, there exists another important, more qualitative, approach whose aim is to discover *which* is the information that flows in order to *declassify* it for guaranteeing non-interference. *Declassifying* information means downgrading the sensitivity of data in order to accommodate with (intentional) information leakage[3]. Robust declassification has been introduced in [118] as a systematic method to drive declassification by characterizing what information flows from confidential to public variables. In particular, the observational attacker's capability is modeled by using equivalence relations as in PER models, and declassification of private data is obtained by manipulating these relations in a semantic-driven way. The semantics considered is the operational semantics, defined on a transition system. The idea is to consider *views* of the computational traces determined by the *observational capability* of the attacker. Hence, given a trace τ of computations of the system S, and given the $\approx$-view of τ (where $\approx$ is an equivalence relation), a view of τ is $\tau/\!\approx$ defined as follows: $\forall i < |\tau|\,.\,(\tau/\!\approx)_i = [\tau_i]_\approx$. The intuition is that a passive attacker (that cannot modify computations), who is

[1] Recall that, given a random variable X, let x ranges over the set of values which X may take and let $p(x)$ the probability that X take x, then $\mathcal{H}(X) = \Sigma_x p(x) \log \frac{1}{p(x)}$. The conditional entropy measuring the uncertainty in the variable X given the knowledge of the variable Y is $\mathcal{H}(X|Y) = \mathcal{H}(X,Y) - \mathcal{H}(X)$.

[2] Recall that, given the random variables X, Y and Z, the conditional mutual information between X and Y given the knowledge of Z is defined as $\mathcal{I}(X;Y|Z) = \mathcal{H}(X|Z) + \mathcal{H}(Y|Z) - \mathcal{H}(X,Y|Z)$.

[3] Note that this is similar to the Cohen's notion of selective dependency [19].

able to distinguish states up to $\approx$, will see the trace τ as a sequence of equivalence classes. Then, an *observation* of the system S, with respect to starting state σ and view $\approx$, is defined as: $Obs_\sigma(S,\approx) \overset{\text{def}}{=} \{\, \tau/\!\approx \,|\, \tau \text{ trace of } S \text{ starting in } \sigma \,\}$. This is the set of all the possible sequences of equivalence classes under $\approx$, that might be observed by watching the system whenever it starts in state σ. At this point, the information that might be learned by observing S through the view $\approx$ is obtained by transforming $\approx$, in function of the set $Obs_\sigma(S,\approx)$. In particular, the authors define a new equivalence relation $S[\approx]$, called *observational equivalence*, such that two states are equivalent only if the possible traces leading from these states are indistinguishable under $\approx$:

$$\forall \sigma, \sigma' \in \Sigma \,.\, \langle \sigma, \sigma' \rangle \in S[\approx] \;\Leftrightarrow\; Obs_\sigma(S,\approx) \equiv Obs_{\sigma'}(S,\approx)$$

Hence, in the paper a system is said secure if all the $\approx$-equivalent states are observationally equivalent. In other words, there is no information flow to an observer with view $\approx$. This characterization is then used in order to declassify data in the system. The basic idea of declassification is that any system that leaks information can be thought of as containing declassification. A passive attacker may be able to learn some information by observing the system but, by assumption, that information leakage is allowed by the security policy [118]. In this way, the attacker is made *blind*, i.e., all the the information that the attacker can get from the execution of the program is declassified. Note that, in [118], robust declassification is defined in the more general case where the attacker can be *active*, namely it can interfere in the execution, for example being a program running concurrently. This work has been recently generalized in [96] in three ways. First, it is shown how to express the property in a language-based setting, for a simple imperative language. Second, the property has been generalized so that untrusted code and data are explicitly part of the system rather than appearing only when there is an active attacker. Third, a security guarantee, called *qualified robustness* has been introduced. This provides untrusted code with a limited ability to affect information release. The key point of this paper is the proof that both robust and qualified declassification can be enforced by a compile-time program analysis based on a simple type system.

More recently, explicit declassification is allowed by weakening the notion of non-interference, in particular in [103] the notion of *delimited information release* is introduced in order to type as secure also systems that admit explicit confidential information release. The idea behind this notion is that a given program is secure as long as updates to variables that are later declassified occur in a way that does not increase the information visible by the attacker [103]. In order to solve the same problem, in [83] the authors define the notion of *relaxed noninterference* . The basic idea is to treat downgrading policies as security levels in traditional information flow systems. Instead of having only two security classes, i.e., `H` and `L` the authors consider a much richer lattice of security levels where each point corresponds to a downgrading policy, describing how the data can be downgraded from this level. Afterwards, the authors define a type system for enforcing the new notion of non-interference.

5.5.2 Constraining attackers

As noted before, the notion of non-interference introduced in this chapter, is based on the assumption that an attacker is able to observe public data, without any observational or complexity restriction. In particular, for some computational systems, disclose any kind of confidential properties require a particular number of statistical tests [41], or a particular computational complexity [82]. The idea is to characterize, in some ways, which has to be the power of the attacker that can disclose certain confidential properties form a given program.

A probabilistic approach.

The notion of non-interference is based on the concept of *indistinguishability* of behaviours: In order to establish that there is no information flow between two objects A and B, it is sufficient to establish that, for any pair of behaviours of the system that differ only in A's object, B's observations cannot distinguish these two behaviours. This suggest that it is possible to weaken this notion by *approximating* this indistinguishability relation [41]. In this paper, the authors replace the notion of indistinguishability by the notion of *similarity.* Therefore, two behaviours, though distinguishable, might still be considered as *effectively* non-interfering, provided that they are similar, i.e., their difference is below a threshold ϵ. A similarity relation can be defined by means of an appropriate notion of distance and provides information on how much two behaviours differ from each other. The power of the attacker is then measured since this quantitative measure of differences between behaviours is related with the number of statistical tests needed to distinguish the two behaviours.

A complexity-based approach.

As noted above, the standard notion of non-interference requires that the public output of the program do not contain *any* information (in the information-theoretic sense) about the confidential inputs. This corresponds to an *all-powerful* attacker who, in his quest to obtain confidential information, has no bounds on the resources (time and space) that it can use. Furthermore, in these definitions an "attacker" is represented by an arbitrary function, which does not even have to be a computable function; the attacker is permitted essentially arbitrary power [82]. The observation made in this paper is that, instead, realistic adversaries are bounded in the resources that they can use. For this reason the author provides a definition of secure information flow that corresponds to an adversary working in probabilistic polynomial time, together with a program analysis that allows to certify these kinds of information flows.