On Extracting Security Policies From Program Invariants

A static analysis for dynamic decision making

José Faustino Fragoso Ana Almeida Matos

February 17, 2011

Abstract

This report approaches the problem of verifying program compliance with information flow policies by proposing a framework that enables dealing with complex and dynamic policies in an efficient and flexible manner. We introduce a calculus for extracting the fundamental dependencies that are encoded into a program which is proved to be both sound and optimal. From the output of this analysis, the strictest security policy under which a program may be executed is then statically inferred. This policy can be used to dynamically decide whether a program is allowed to run, or as a comprehensive and succinct digest of the reasons for which a program is not deemed secure.

Contents

1 Introduction

Information flow security regards the compliance of a program to a policy that restricts what information can be inferred from the execution of the program by a given observer of the system. In the classical setting [18], the program is analysed against a certain lattice of security levels, and typically suffers from imprecision and inflexibility problems, if the analysis is done statically, or soundness and efficiency problems, if it is done dynamically. This report proposes a different strategy to information flow analysis, aiming to profit from the efficiency of statical approaches, and the flexibility of the dynamic ones.

The main observation that motivates the proposal is that information flows that are encoded in a programs are not intrinsically secure or insecure, but are so with respect to a given context. Context, which includes the relevant security policy and security labellings, is dynamic by nature. For this reason, classical approaches to information flow analysis that rely on a predefined security labelling of levels that are related by a single flow policy are often inadequate for real world applications.

Declassification mechanisms offer different degrees and forms of flexibility to an information flow analysis [19]. They enable dynamic security labellings [22], dynamic policies [3, 6] and also deal with differences in policies that arise in distributed contexts [2]. However, even when incorporating such mechanisms, a framework that statically accepts or rejects programs is inherently constrained.

In a scenario in which access permissions (or security labellings) that are attached to resources are allowed to change, compliance of a given program with a given information flow policy depends on the runtime labelings of the resources that the program handles. Complementarily, the security policies that establish which information flows are allowed in a system may also vary over time or according to the computational domain in which the program is executed. It is therefore desirable to postpone the decision of whether a program is allowed to run to as close to runtime as possible. Clearly, this poses an additional overhead on program execution (which can be quite considerable depending on the size of the program to execute).

This report proposes a mechanism for coping with dynamic and distributed security labelings, as well as with dynamic and distributed security policies, that relies on a static dependency analysis between the computational objects of a program. The aim is to infer the information flows entailed by the program in order to perform sound and dynamic decisions concerning the execution of the program in an efficient manner.

More precisely, the main technical contributions of this report are:

- A calculus of dependencies that is precise with respect to a classical type system, in the sense that verifying typability of a program is equivalent to verifying that variables only depend (syntactically) on variables that are lower with respect to a given typing security labelling.
- A method for determining, for a given program and type system, the weakest security labelling that renders the program typable.
- A mechanism for calculating the strictest flow policy with which a program complies, when fixing the security labelling.

- A discussion of several different ways to express security policies over sets of principals.

The above technical results can be applied to build a platform for making dynamic decisions regarding program security, that is adequate in the sense of correctness, efficiency and practicality. Furthermore, we point out its usefulness in building tools for assisting the development of secure programs. More concretely, the following direct applications of the technical contributions in this report may be highlighted:

- The strictest flow policy entailed by a program can be used for efficiently verifying compliance with a given security policy (be it a confidentiality or integrity policy). This can be applied in a variety of contexts, ranging from access control runtime monitors to proof carrying code.
- The strictest flow policy provides a succinct and comprehensive explanation of the reasons why the program is not considered secure. Thus, it can be used as an error-reporting mechanism.
- At the programming level, the specification of security labellings can be left incomplete, leading to the computation of the weakest labelling higher than the specified one which types the program. This inference mechanism is different from traditional type inference for information flow because the inference procedure is allowed to raise the security level of any variable. Nevertheless, it always yields a correct labelling.

1.1 Synopsis

We consider a setting in which (integrity and confidentiality) security levels are sets of principals [3, 16], similar to access control lists. In such a scenario security lattices correspond to closure operators over the powerset of the set of principals. Section 2 covers this topic in detail and presents several illustrative examples. Section 3 states the security property that is considered in this report and instantiates the Volpano *et al.* type system to enforce both confidentiality and integrity policies expressed as closure operators. Section 4 presents an analysis for computing the strictest integrity policy with which a program complies. This analysis is specially important because, when taking the program variables as the principals of the system, it yields a dependency calculus (Section 5) which is the cornerstone of all remaining analysis. This dependency calculus is then use in two ways: in Section 7 it is used to infer the lowest security labelling with which a program complies (when fixing the security lattice), whereas in Section 8 it is used to infer the lowest confidentiality policy with which a program complies (when fixing the labelling).

2 Specifying Confidentiality Policies

Originally [9] the problem of certifying that a given program observes a given confidentiality policy was formulated in the following way: one has to specify a security lattice over a set of confidentiality levels and a security labelling which assigns to

each variable a confidentiality level and the goal is to guarantee that information pertaining to high confidentiality levels does not leak to low confidentiality levels. The *non-interference* property [10] provides a formal meaning for information leakage.

In information flow research the term *security lattice* is pervasive, whereas its meaning may actually differ according to the author in question. In this report, we shall abide by the notion that a security lattice is always generated by a closure operator over a given set of security principals [3, 25], as it is carefully explained below.

Given a set P of *principals*, a *confidentiality level* is any subset l of P. From this point of view, the confidentiality level P is the most public one (also denoted $\bot$): a variable labeled P may be read by every principal. Conversely, $\varnothing$ is the most secret security level (also denoted $\top$): a variable labelled $\varnothing$ is so secret that no one can read it. In this setting, we can interpret reverse inclusion of security levels as indicating allowed flows of information: if a variable x is labelled l and $l \supseteq l'$, then the value of x may be transferred to a variable y labelled l'.

2.1 Security lattices generated *via* flow relations

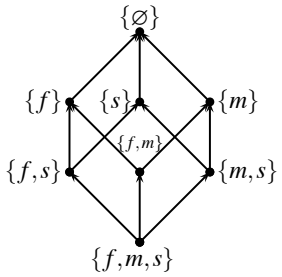
A *flow relation* is a binary relation over P. A pair $(p,q) \in F$ is to be understood as "information may flow from principal p to principal q". That is: "everything that principal p is allowed to read may also be read by principal q". Naturally, every flow relation over P induces a closure operator over $\mathcal{P}(P)$:

$$L\uparrow_F = \{l' \mid \exists l \in L \cdot l F^* l'\} \tag{1}$$

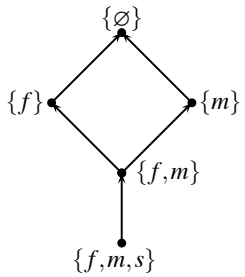
where F^* is to be understood as the reflexive-transitive closure of F.

Since $\uparrow_F$ is a closure operator, its range is a Moore family and thus a lattice when equipped with set inclusion as its partial order. This means that when establishing a flow relation over a given set of principals, we are implicitly reducing the size of the original security lattice, as illustrated in the following example.

Example 1. *Consider the following security lattice:*



Here, the set of pricipals is $\{f,m,s\}$ where f, m and s stand for father, mother and son respectively. Now suppose that the members of this family wish to implement a security policy such that everything that the son can read may also be read by his parents. This policy can be stated as: $F = \{(s,f),(s,m)\}$. Having established this flow relation, when assigning the security level s to a given variable x, one is in fact assigning security level $\{s\}\uparrow_F = \{f,m,s\}$. As such, the new security lattice will be:



The previous example shows that when establishing a flow relation, one is indeed collapsing confidentiality levels in $\mathcal{P}(P)$, thus obtaining a new lattice that shall be considered the *de facto* security lattice.

As it is, we are now considering security lattices generated by closure operators which are in turn generated by arbitrary binary relations (flow relations) over a finite set. However, when generating a closure operator through a flow relation, one ensures that it is actually a *additive closure operator* which is proved in the following lemma.

Lemma 1. *Given a flow relation F over a finite set of principals P, the closure operator $\uparrow_F$ is an additive closure operator, that is:*

$$(l_1 \cup l_2)\uparrow_F = l_1\uparrow_F \cup l_2\uparrow_F$$

Proof. From monotonocity of the closure operator it follows that:

$$l_1\uparrow_F \cup l_2\uparrow_F \subseteq (l_1 \cup l_2)\uparrow_F$$

For the converse inclusion, observe that if $p \in (l_1 \cup l_2)\uparrow_F$, this means that there must exist $q \in l_1 \cup l_2$ such that qF^*p. However, q must belong to either to l_1 or to l_2, so $p \in l_1\uparrow_F$ or $p \in l_2\uparrow_F$. □

So each security lattice obtained by establishing a flow relation F over a given finite set of principals P corresponds to an additive closure operator $\uparrow_F$. Naturally, each additive closure operator over $\mathcal{P}(P)$ is completely defined once defined for each element of P since:

$$l\uparrow_F = \bigcup_{p\in l}\{p\}\uparrow_F \tag{2}$$

Therefore, we establish the notion of *flow function*, that is, a function which assigns to each principal p the set of principals that can read all the information labeled with p. For instance, if $q \in \pi(p)$, then principal q may read all the information that principal p can read. Every flow function $\pi : P \to \mathcal{P}(P)$ must verify the following two properties:

$$\begin{array}{l} \forall p \in P \cdot p \in \pi(p) \\ \forall p,q \in P \cdot q \in \pi(p) \Rightarrow \pi(q) \subseteq \pi(p) \end{array} \tag{3}$$

Given a flow relation F, its corresponding flow function can be obtained in the following way:

$$\pi_F(p) = \{p\}\uparrow_F \tag{4}$$

Given a flow function $\pi : P \to \mathcal{P}(P)$, define the operator $\pi^* : \mathcal{P} \to \mathcal{P}(P)$ as:

$$\pi^*(l) = \bigcup_{p \in l} \pi(p) \tag{5}$$

Lemmas 2 and 3 establish the relation between flow functions as defined in equation 3 and closure operators.

Lemma 2. *If π is a flow function then π^* (defined as above) is an additive closure operator.*

Proof.

- Monotonocity: $l_1 \subseteq l_2 \Rightarrow \pi^*(l_1) \subseteq \pi^*(l_2)$
  Assuming that $l_1 \subseteq l_2$.

$$\begin{gathered} \pi^*(l_1) = \cup_{p \in l_1} \pi(p) \\ \cup_{p \in l_1} \pi(p) \subseteq \cup_{p \in l_2} \pi(p) \text{ since } l_1 \subseteq l_2 \\ \pi^*(l_1) \subseteq \pi^*(l_2) \end{gathered}$$

- Extensivity: $l_1 \subseteq \pi^*(l_1)$; since π is a flow function, for every principal: $p \in \pi(p)$.
- Idempontence: $\pi^*(\pi^*(l_1)) = \pi^*(l_1)$. We only have to prove that $\pi^*(\pi^*(l_1)) \subseteq \pi^*(l_1)$, the converse inclusion comes from extensivity. So let's assume that $p \in \pi^*(\pi^*(l_1))$. This implies that there is a $q \in \pi^*(l_1)$ such that $p \in \pi(q)$. On the other hand, the fact that $q \in \pi^*(l_1)$, implies that there is a $s \in l_1$ such that $q \in \pi(s)$. So, assuming that $p \in \pi^*(\pi^*(l_1))$, we must conclude that there are $q, s \in P$ such that: $s \in l_1$, $q \in \pi(s)$ and $p \in \pi(q)$. But, since π is a flow function, we must conclude that $p \in \pi * (l_1)$.

□

Lemma 3. *If $\Pi : \mathcal{P}(P) \to \mathcal{P}(P)$ is a closure operator then the function $\pi : P \to \mathcal{P}(P)$ such that $\pi(p) = \Pi(\{p\})$ is a flow function.*

Proof.

- $\pi(p) = \Pi(\{p\})$. Applying extensivity: $p \in \pi(p)$.
- Suppose $q \in \pi(p)$. Then: $\{q\} \subseteq \Pi(\{p\})$. By monotonocity: $\Pi(\{q\}) \subseteq \Pi(\Pi(\{p\}))$. Then, applying idempotence: $\Pi(\{q\}) \subseteq \Pi(\{p\})$.

□

Clearly, given a finite set of principles P there is a one-to-one correspondence between the set of flow functions over P and the set of security lattices generated by the flow relations over P. Given a flow function $\pi : P \to \mathcal{P}(P)$, the confidentiality lattice corresponding to π shall be denoted:

$$\mathcal{L}_\pi = (\pi^*(\mathcal{P}(P)), \supseteq) \tag{6}$$

The order on flow functions (and thus on security lattices generated by flow functions) is defined as follows:

$$\pi_1 \sqsubseteq \pi_2 \Leftrightarrow \forall_{p \in P}.\pi_1(p) \subseteq \pi_2(p) \tag{7}$$

The set of flow functions over a given set of principals P equipped with the order introduced above is a lattice as the following lemma proves.

Lemma 4. *When equipped with the ordering introduced in 7, the set of flow functions over a finite set of principals P is a lattice.*

Proof. It is clear that it is a poset. It remains to show that it is also a lattice. To accomplish this we propose the following greatest lower bound:

$$(\pi_1 \sqcap \pi_2)p = \pi_1(p) \cap \pi_2(p)$$

having to prove that the lower bound proposed is itself a flow function.

- Assuming that π_1 and π_2 are flow functions, it follows that for every principal p, $p \in \pi_1(p)$ and $p \in \pi_2(p)$ and thus $p \in (\pi_1 \sqcap \pi_2)(p)$.
- Suppose $p \in (\pi_1 \sqcap \pi_2)(q)$, by definition it must be the case that $p \in \pi_1(q)$ and $p \in \pi_2(q)$. Since, π_1 and π_2 are assumed to be flow functions, it follows that $(\pi_1 \sqcap \pi_2)(p) \subseteq (\pi_1 \sqcap \pi_2)(q)$.

□

The order on flow functions introduced above may be understood in the following way: if $\pi_1 \sqsubseteq \pi_2$, then π_1 is more strict than π_2 in the sense that not all the information flows allowed by π_2 are allowed by π_1. However, all the information flows allowed by π_1 must also be allowed by π_2. Otherwise, they would not be comparable. The following lemma clarifies this interpretation concerning the order relation established on flow functions.

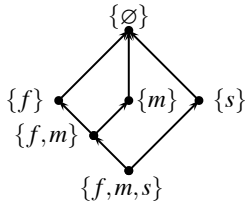
Lemma 5. *For any two flow functions π_1 and π_2 over a given set of principals P, if $\pi_1 \sqsubseteq \pi_2$ then for any security levels $l_1, l_2 \in \mathcal{P}(P)$, if $\pi_1^*(l_1) \subseteq \pi_1^*(l_2)$, then $\pi_2^*(l_1) \subseteq \pi_2^*(l_2)$.*

Proof.
By extensivity: $l_1 \subseteq \pi_1^*(l_1)$.
Applying the hypothesis: $l_1 \subseteq \pi_1^*(l_2)$
By monotonocity: $\pi_2^*(l_1) \subseteq \pi_2^*(\pi_1^*(l_2))$
By hypothesis: $\pi_1^*(l_2) \subseteq \pi_2^*(l_2)$
By monotonocity and extensivity: $\pi_2^*(\pi_1^*(l_2)) \subseteq \pi_2^*(l_2)$ □

2.2 Security lattices generated by arbitrary closure operators

Section 2.1 states that the security lattices that arise from establishing flow relations over a given set of principals P correspond to those lattices generated by the additive closure operators over $\mathcal{P}(P)$. However, there are confidentiality policies which may not be captured by the security lattices generated in that way as is illustrated in the following example.

Example 2. *Suppose that the family considered in example 1 wishes to change its confidentiality policy so that all of the son's information that is shown to one parent is also shown to the other. It is not possible to enforce this particular confidentiality policy through a flow relation. To specify this confidentiality policy one has to specify the closure operator itself. Furthermore, this closure operator is not additive.*



Example 2 stresses the fact that in some situations flow relations are not expressive enough to model the confidentiality policy which one wishes to enforce. In such situations the closure operator must be fully specified. Therefore, flow functions cease to be useful, in this context the term *flow operator* shall be used.

As done for flow functions, one can establish an order on closure operators over $\mathcal{P}(P)$ in a similar way:

$$\Pi_1 \sqsubseteq \Pi_2 \Leftrightarrow \forall_{l \in \mathcal{P}(P)} \,.\, \Pi_1(l) \subseteq \Pi_2(l) \tag{8}$$

thus obtaining a lattice as proved in lemma 6.

Lemma 6. *The set of closure operators over $\mathcal{P}(P)$ equipped with the order defined in 8 is a lattice.*

Lemma 7. *Given two closure operators over $\mathcal{P}(P)$, Π_1 and Π_2, and two security levels $l_1, l_2 \in \mathcal{P}(P)$, such that $\Pi_1 \sqsubseteq \Pi_2$ and $\Pi_1(l_1) \subseteq \Pi_1(l_2)$, then $\Pi_2(l_1) \subseteq \Pi_2(l_2)$.*

The proofs of lemmas 6 and 7 shall be omitted since they are similar to the proofs of the corresponding lemmas in the preceeding section: 5 and 4.

Given a set of principals P and a closure operator $\Pi : \mathcal{P}(P) \to \mathcal{P}(P)$ over $\mathcal{P}(P)$, the confidentiality lattice generated by Π is:

$$\mathcal{L}_\Pi = (\Pi(\mathcal{P}(P)), \supseteq) \tag{9}$$

2.3 Specifying Integrity Policies

Confidentiality policies constrain *who* can read the secret data and therefore where the secret data can flow to, hence the notion of *flow function* which assigns to each principal the set of principals who can read his/her information. Complementarily, one can assign to each principal the set of principals who 'trust him to be a reliable source'. This kind of policy is deemed an *integrity policy* [5]. That is, an integrity policy constrain who can write the data and thus where the data may have come from.

Again, integrity policies shall be expressed as closure operators over a given set of security principals. Given a set P of principals, an integrity level is any subset l of P.

Thus, the integrity level P corresponds to those variables who can be written by every principal, whereas $\varnothing$ corresponds to those variables that no one can change.

Clearly, flow functions can be used to specify integrity policies. However, they must be interpreted in a complementary way: if $q \in \pi(p)$, then principal q is allowed to change information belonging to principal p and therefore, in this sense, information belonging to principal p does also belong to principal q. Analogously, if instead of considering flow functions, one considers flow operators the same interpretation holds.

For instance, when interpreted as an integrity policy, the flow policy stated in example 1 settles that the parents can change all information belonging to their son (all the information that belongs to the son also belongs to his parents). Similarly, taken as an integrity policy, the flow operator stated in example 2 establishes that all the information belonging to the son which is mother can change may also be changed by his father and *vice-versa*.

As observed in [14], "this style of integrity policy can be defined formally using the same definition of noninterference used for confidentiality". However, *using the same defininition of noninterference* for confidentiality and integrity, requires ordering integrity levels under subset inclusion (and not under reverse subset inclusion as happened with confidentiality levels). Therefore, given an integrity policy generated by a closure operator $\Pi : \mathcal{P}(P) \to \mathcal{P}(P)$, the lattice of integrity levels generated by Π is defined to be:

$$\mathcal{L}_{\Pi} = (\Pi(\mathcal{P}(P)), \subseteq) \tag{10}$$

3 Trace Noninterference

Vopano *et al* introduced in [23] a type system for enforcing non-interference. However, as observed in [23], this type system enforces a much stronger property than plain noninterference [10]. This particular version of noninterference shall be henceforth referred to as *trace noninterference*.

This section presents a formal definition of trace noninterference (in definition 5), as well as the usual type system for enforcing trace noninterference (in definition 6). Both are specified for arbitrary security lattices. As such, both can be instantiated for confidentiality and integrity lattices generated by closure operators as discussed in previous sections.

This report uses a simple `WHILE` language (presented in definition 1). Its semantics (presented in definition 2), though fairly standard, is borrowed from [25].

Definition 1 (Model Language)**.**

$$\begin{aligned} c \ ::= & \ \text{skip} \mid \text{while } b \text{ do } c \mid v := e \mid \\ & \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid c_1 ; c_2 \end{aligned}$$

Definition 2 (Operational Semantics)**.**

$$
\begin{array}{rcll}
< v := e, M > & \xrightarrow{v} & < skip, M[v \mapsto M(e)] > & \\
< if\ b\ then\ c_1\ else\ c_2, M > & \xrightarrow{\cdot} & < c_1, M > & if\ M(b) = true \\
< if\ b\ then\ c_1\ else\ c_2, M > & \xrightarrow{\cdot} & < c_2, M > & if\ M(b) = false \\
< while\ b\ do\ c, M > & \xrightarrow{\cdot} & < c; while\ b\ do\ c, M > & if\ M(b) = true \\
< while\ b\ do\ c, M > & \xrightarrow{\cdot} & < skip, M > & if\ M(b) = false \\
< skip; c_2, M > & \xrightarrow{\alpha} & < c_2, M > & \\
< c_1; c_2, M > & \xrightarrow{\alpha} & < c_1'; c_2, M' > & if < c_1, M > \xrightarrow{\alpha} < c_1', M' >
\end{array}
$$

The operational semantics defined in 2 is a structural operational semantics. Each configuration $< c, M >$ consists of a program c and a memory M. Each transition $< c, M > \xrightarrow{\alpha} < c', M' >$ from configuration $< c, M >$ to configuration $< c', M' >$ is annotated with an *event* α, which is either a $\cdot$, indicating that no assignment occurred during the corresponding transition step, or a given variable v, indicating that v has been updated. Terminal configurations have the form $< skip, M >$.

The relation $\rightarrow$ is obtained from $\xrightarrow{\alpha}$ by erasing the annotated event. A configuration $< c, M >$ is said to converge, if there is a memory M' such that $< c, M > \rightarrow^* < skip, M' >$, written $< c, M > \Downarrow M'$, where $\rightarrow^*$ denotes the reflexive transitive closure of $\rightarrow$, otherwise it is said to diverge. When M' is unimportant it can be omitted, thus writing $< c, M > \Downarrow$.

The trace of the execution of configuration $< c, M >$, denoted $Trace(< c, M >)$, is the sequence of configurations:

$$< c_0, M_0 > \xrightarrow{\alpha_0} < c_1, M_1 > \xrightarrow{\alpha_1} \cdots < c_i, M_i > \xrightarrow{\alpha_i} \cdots \tag{11}$$

Definition 3 (l-projection of a trace)**.** *For any trace t, given an arbitrary security lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$, a security labelling $\Gamma : Var \rightarrow L$ and a security level $l \in L$, the l-projection of the trace t, written $t|_l^{\Gamma,\mathcal{L}}$, where:*

$$t = < c_0, M_0 > \xrightarrow{\alpha_0} < c_1, M_1 > \xrightarrow{\alpha_1} \cdots < c_i, M_i > \xrightarrow{\alpha_i} \cdots$$

is the sequence of (l-restrictions), of memories:

$$m = M_0|_l^{\Gamma,\mathcal{L}}, M_{i_1}|_l^{\Gamma,\mathcal{L}}, M_{i_2}|_l^{\Gamma,\mathcal{L}}, \cdots$$

such that $0 < i_1 < i_2 < \cdots$ and for every transition $< c_j, M_j > \xrightarrow{\alpha_j} < c_{j+1}, M_{j+1} >$ in t, if α_j corresponds t a variable and $\Gamma(\alpha_j) \sqsubseteq l$, then $j + 1 = i_k$ for some k, and for every i_k in m there is a j such that $j + 1 = i_k$.

In definition 3, the restriction $M|_l^{\Gamma,\mathcal{L}}$ of memory M to security level l with respect to security labelling Γ is defined by restricting the mapping (correspondign to memory M) to variables whose security level is at or below l.

If two memories (or traces) are equal at a certain security level $l \in L$, they are said to be *indistinguishable* at level l. Accordingly, definition 4 lifts indistinguishability at given level l from memories (and thus traces) to configurations. Notice that *weak indistinguishability* is termination insensitive because it deems a diverging configuration weakly indistinguishable from any other. *Trace noninterference* is introduced in defintion 5. Informally, a program is said to verify trace noninterference if, given two

memories indistinguishable at level l, the execution traces corresponding to execution of the program on each memory are also weakly indistinguishable up to l.

Definition 4 (Weak Indistinguishability)**.** *Given a security lattice $\mathcal{L}$ and a security labelling $\Gamma : Var \to L$, two confiturations $< c_1, M_1 >$ and $< c_2, M_2 >$ are said to be weakly indistinguishable up to the security level l (written $< c_1, M_1 > \approx_l^{\Gamma,\mathcal{L}} < c_2, M_2 >$) if whenever their corresponding traces ($t_1 = Trace(< c_1, M_1 >)$ and $t_2 = Trace(< c_2, M_2 >)$) terminate:*

$$t_1|_l^{\Gamma,\mathcal{L}} = t_2|_l^{\Gamma,\mathcal{L}}$$

Definition 5 (Trace non-interence)**.** *Given a security lattice, $\mathcal{L} = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \top, \bot)$, a security labelling $\Gamma : Var \to L$ and a program c, we say that c verifies non-interference with respect to Γ and $\mathcal{L}$, which is denoted by $\Gamma \vDash_{\mathcal{L}} c$, if:*

$$\forall l, M_1, M_2 . M_1 =_l M_2 \Rightarrow < c, M_1 > \approx_l^{\Gamma,\mathcal{L}} < c, M_2 >$$

3.1 A type system for trace noninterference

Definition 6 presents the classical type system for enforcing noninterference [23]. Each typing judgement has the form $\Gamma \vdash_{\mathcal{L}} c : l$ where l corresponds to the writing effect of c, that is: the greatest lower bound on the security level of the variables updated in c.

Definition 6 (Volpano-Smith-Irvine Type System)**.**

SKIP
$$\frac{}{\Gamma \vdash_{\mathcal{L}} \texttt{skip} : \top}$$

ASSIGN
$$\frac{l_e \sqsubseteq \Gamma(x) \quad l_e = \bigsqcap_{y \in vars(e)} \Gamma(y)}{\Gamma \vdash_{\mathcal{L}} \texttt{x := e} : \Gamma(x)}$$

SEQ
$$\frac{\Gamma \vdash_{\Pi} \texttt{c}_1 : l_1 \quad \Gamma \vdash_{\Pi} \texttt{c}_2 : l_2}{\Gamma \vdash_{\Pi} \texttt{c}_1\texttt{;c}_2 : l_1 \sqcap l_2}$$

IF
$$\frac{l_b = \bigsqcap_{y \in vars(b)} \Gamma(y) \quad \Gamma(b) \sqsubseteq l_1 \sqcap l_2 \quad \Gamma \vdash_{\mathcal{L}} \texttt{c}_1 : l_1 \quad \Gamma \vdash_{\mathcal{L}} \texttt{c}_2 : l_2}{\Gamma \vdash_{\mathcal{L}} \texttt{if b then c}_1 \texttt{ else c}_2 : l_1 \sqcap l_2}$$

WHILE
$$\frac{\Gamma \vdash_{\mathcal{L}} \texttt{c} : l \quad l_b = \bigsqcap_{y \in vars(b)} \Gamma(y) \quad l_b \sqsubseteq l}{\Gamma \vdash_{\mathcal{L}} \texttt{while b do c} : l}$$

Lemma 8 (Soundness of the Type System)**.** *Given a security lattice, $\mathcal{L} = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \top, \bot)$, a security labelling $\Gamma : Var \to L$ and a program c, if there is a security level $l \in L$ such that $\Gamma \vdash_{\mathcal{L}} c : l$ then $\Gamma \vDash_{\mathcal{L}} c$.*

Lemma 8 states that the Volpano *et al* type system is sound with respect to trace noninterference. We shall omit the proof of 8, the interested reader is referred to [25].

3.2 Typing programs by comparing policies

The type system stated in definition 6 can be instantiated in order to certify a given program with respect to a given integrity policy (specified by a closure operator Π_I) and a given confidentiality policy (specified by a closure operator Π_C), which is illustrated in definition 7. In this case, the security lattice corresponds to the cartesian product of the integrity lattice and the confidentiality lattice:

$$\mathcal{L} = \mathcal{L}_{\Pi_I} \times \mathcal{L}_{\Pi_C} = (\Pi_I(\mathcal{P}(P)) \times \Pi_C(\mathcal{P}(P)), (\subseteq, \supseteq)) \tag{12}$$

Definition 7 (Volpano-Smith-Irvine type system for certifying confidentiality and integrity)**.**

SKIP
$$\frac{}{\Gamma \vdash_{\mathcal{L}_{\Pi_I} \times \mathcal{L}_{\Pi_C}} \texttt{skip} : (P, \varnothing)}$$

ASSIGN
$$\frac{\begin{array}{c}\Pi_I(\bigcup_{v \in vars(e)} \Gamma(v)) \subseteq \Pi_I(\Gamma(x)) \\ \Pi_C(\Gamma(x)) \subseteq \bigcap_{v \in vars(e)} \Pi_C(\Gamma(v))\end{array}}{\Gamma \vdash_{\mathcal{L}_{\Pi_I} \times \mathcal{L}_{\Pi_C}} : (\Pi_I(\Gamma(x)), \Pi_C(\Gamma(x)))}$$

SEQ
$$\frac{\begin{array}{c}\Gamma \vdash_{\mathcal{L}_{\Pi_I} \times \mathcal{L}_{\Pi_C}} c_1 : (l_I^1, l_C^1) \\ \Gamma \vdash_{\mathcal{L}_{\Pi_I} \times \mathcal{L}_{\Pi_C}} c_2 : (l_I^2, l_C^2)\end{array}}{\Gamma \vdash_{\mathcal{L}_{\Pi_I} \times \mathcal{L}_{\Pi_C}} c_1; c_2 : (l_I^1 \cap l_I^2, \Pi_C(l_C^1 \cup l_C^2))}$$

IF
$$\frac{\begin{array}{ll}\Pi_I(\bigcup_{v \in varsb(b)} \Gamma(v)) \subseteq l_I^1 \cap l_I^2 & \Gamma \vdash_{\mathcal{L}_{\Pi_I} \times \mathcal{L}_{\Pi_C}} c_1 : (l_I^1, l_C^1) \\ \Pi_C(l_C^1 \cup l_C^2) \subseteq \bigcap_{v \in vars(b)} \Pi_C(\Gamma(v)) & \Gamma \vdash_{\mathcal{L}_{\Pi_I} \times \mathcal{L}_{\Pi_C}} c_2 : (l_I^2, l_C^2)\end{array}}{\Gamma \vdash_{\mathcal{L}_{\Pi_I} \times \mathcal{L}_{\Pi_C}} \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 : (l_I^1 \cap l_I^2, \Pi_C(l_C^1 \cup l_C^2))}$$

WHILE
$$\frac{\begin{array}{ll}\Gamma \vdash_{\mathcal{L}} c : (l_I, l_C) & \Pi_I(\bigcup_{v \in vars(b)} \Gamma(v)) \subseteq l_I \\ & l_C \subseteq \bigcap_{v \in vars(b)} \Pi_C(\Gamma(v))\end{array}}{\Gamma \vdash_{\mathcal{L}_{\Pi_I} \times \mathcal{L}_{\Pi_C}} \texttt{while } b \texttt{ do } c : (l_I, l_C)}$$

It is worth noting that when considering a scenario in which the security policies under which a program is executed are determined by the site where the program is executed rather than by the programmer (a typical Web scenario) and in which a program may be executed several times, it is much more useful to compute the strictest security policies to which the program complies (or to verify them, if they are provided with the program) rather than just checking if the program is compliant with the policy that one wishes to enforce. The first approach allows the site to change its allowed policies without having to reanalyse all the programs that were already so. Thus, it only has to verify for each program if its corresponding policies are stricter than the new ones the domain wishes to enforce.

Lemmas 9 and 10 prove that once found the closure operators corresponding to the strictest integrity and confidentiality lattices to which a program complies, the problem of verifying if this program is compliant with a given security policy (be it an integrity or a confidentiality policy) amounts to a comparison between closure operators.

Lemma 9. *For every program c, security labelling* $\Gamma : Var \to \mathcal{P}(P)$*, and two confidentiality policies generated by the closure operators* Π^1_C, Π^2_C *respectively, such that* $\Pi^1_C \sqsubseteq \Pi^2_C$*, if there is a security level* $l_1 \subseteq P$ *such that:*

$$\Pi^1_C \circ \Gamma \vdash_{\mathcal{L}_{\Pi^1_C}} c : l_1$$

Then, there is a security level $l_2 \subseteq P$ *such that:*

$$\forall_{v \in Var} \,.\, l_1 \subseteq \Pi^1_C(\Gamma(v)) \Rightarrow l_2 \subseteq \Pi^2_C(\Gamma(v)) \;\wedge\; \Pi^2_C \circ \Gamma \vdash_{\mathcal{L}_{\Pi^2_C}} c : l_2$$

Proof. Without loss of generality, the security levels corresponding to the writing effect of the typing judgements are closed with respect to the closure operator associated with the security lattice under which the command is being typed. We proceed by induction on the derivation of $\Pi_1 \circ \Gamma \vdash_{\mathcal{L}_{\Pi^1_C}} c : l_1$.

- **assign:** `x := e`.

  $$\begin{array}{l} \Pi^1_C(\Gamma(x)) \subseteq \bigcap_{v \in vars(e)} \Pi^1_C(\Gamma(v)) \\ \forall_{v \in vars(e)} . \Pi^1_C(\Gamma(x)) \subseteq \Pi^1_C(\Gamma(v)) \\ \forall_{v \in vars(e)} . \Pi^2_C(\Gamma(x)) \subseteq \Pi^2_C(\Gamma(v)) \quad \text{(Applying lemma 7)} \\ \Pi^2_C(\Gamma(x)) \subseteq \bigcap_{v \in vars(e)} \Pi^2_C(\Gamma(v)) \end{array}$$

  Suppose that for a given variable $v \in Var$, $l_1 \subseteq \Pi^1_C(\Gamma(v))$, then:

  $$\begin{array}{l} \Pi^1_C(\Gamma(x)) \subseteq \Pi^1_C(\Gamma(v)) \\ \Pi^2_C(\Gamma(x)) \subseteq \Pi^2_C(\Gamma(v)) \quad \text{(Applying lemma 7)} \\ l_2 \subseteq \Pi^2_C(\Gamma(v)) \end{array}$$

- **if:** `if b then` c_1 `else` c_2. The hypothesis ensures that there are security levels l'_1 and and l''_1 such that:

  $$\begin{array}{c} \Pi^1_C \circ \Gamma \vdash_{\mathcal{L}_{\Pi^1_C}} c_1 : l'_1 \\ \Pi^1_C \circ \Gamma \vdash_{\mathcal{L}_{\Pi^1_C}} c_2 : l''_1 \\ l_1 = \Pi^1_C(l'_1 \cup l''_1) \quad l_1 \subseteq \bigcap_{v \in vars(b)} \Pi^1_C(v) \end{array}$$

  The induction hypothesis guarantees the existence two security levels l'_2 and l''_2 such that:

  $$\begin{array}{l} \forall_{v \in Var} \,.\, l'_1 \subseteq \Pi^1_C(\Gamma(v)) \Rightarrow l'_2 \subseteq \Pi^2_C(\Gamma(v)) \;\wedge\; \Pi^2_C \circ \Gamma \vdash_{\mathcal{L}_{\Pi^2_C}} c_1 : l'_2 \\ \forall_{v \in Var} \,.\, l''_1 \subseteq \Pi^1_C(\Gamma(v)) \Rightarrow l''_2 \subseteq \Pi^2_C(\Gamma(v)) \;\wedge\; \Pi^2_C \circ \Gamma \vdash_{\mathcal{L}_{\Pi^2_C}} c_2 : l''_2 \end{array}$$

  Naturally, $l'_1 \cup l''_1 \subseteq \Pi^1_C(\Gamma(v)) \Rightarrow l'_2 \cup l''_2 \subseteq \Pi^2_C(\Gamma(v))$. Furthermore:

  $$\begin{array}{l} \Pi^1_C(l'_1 \cup l''_1) \subseteq \Pi^1_C(\Gamma(v)) \\ l'_1 \cup l''_1 \subseteq \Pi^1_C(\Gamma(v)) \quad \text{(Applying extensivity)} \\ l'_2 \cup l''_2 \subseteq \Pi^2_C(\Gamma(v)) \\ \Pi^2_C(l'_2 \cup l''_2) \subseteq \Pi^2_C(\Gamma(v)) \quad \text{(Applying monotonocity and extensivity)} \end{array}$$

Thus proving the first claim. The proof of $\Pi^2_C \circ \Gamma \vdash_{\mathcal{L}_{\Pi^2_C}} c : l_2$ follows:

$$\begin{gathered}\Pi^1_C(l'_1 \cup l''_1) \subseteq \bigcap_{v \in vars(b)} \Pi^1_C(\Gamma(v)) \\ \forall_{v \in vars(b)} \Pi^1_C(l'_1 \cup l''_1) \subseteq \Pi^1_C(\Gamma(v)) \\ \forall_{v \in vars(b)} \Pi^2_C(l'_2 \cup l''_2) \subseteq \Pi^2_C(\Gamma(v)) \\ \Pi^2_C(l'_2 \cup l''_2) \subseteq \bigcap_{v \in vars(b)} \Pi^2_C(\Gamma(v))\end{gathered}$$

- **seq:** $\mathtt{c}_1;\mathtt{c}_2$. The hypothesis ensures that there are security levels l'_1 and and l''_1 such that:

$$\begin{gathered}\Pi^1_C \circ \Gamma \vdash_{\mathcal{L}_{\Pi^1_C}} c_1 : l'_1 \\ \Pi^1_C \circ \Gamma \vdash_{\mathcal{L}_{\Pi^1_C}} c_2 : l''_1 \\ l_1 = l'_1 \cap l''_1\end{gathered}$$

  The induction hypothesis guarantees the existence two security levels l'_2 and l''_2 such that:

$$\begin{gathered}\forall_{v \in Var} \,.\, l'_1 \subseteq \Pi^1_C(\Gamma(v)) \Rightarrow l'_2 \subseteq \Pi^2_C(\Gamma(v)) \;\wedge\; \Pi^2_C \circ \Gamma \vdash_{\mathcal{L}_{\Pi^2_C}} c_1 : l'_2 \\ \forall_{v \in Var} \,.\, l''_1 \subseteq \Pi^1_C(\Gamma(v)) \Rightarrow l''_2 \subseteq \Pi^2_C(\Gamma(v)) \;\wedge\; \Pi^2_C \circ \Gamma \vdash_{\mathcal{L}_{\Pi^2_C}} c_2 : l''_2\end{gathered}$$

  The first claim follows exactly in the same way as in the [if] case. The second claim is immediate.

- **while:** `while b do c`. The hypothesis ensures that there is a security level l_1 such that:

$$\Pi^1_C \circ \Gamma \vdash_{\mathcal{L}_{\Pi^1_C}} c : l_1 \quad l_1 \subseteq \bigcap_{v \in vars(b)} \Pi^1_C(\Gamma(v))$$

  The induction hypothesis guarantees the existence of a security level l_2 such that:

$$\begin{gathered}\forall_{v \in Var} \,.\, l_1 \subseteq \Pi^1_C(\Gamma(v)) \Rightarrow l_2 \subseteq \Pi^2_C(\Gamma(v)) \\ \Pi^2_C \circ \Gamma \vdash_{\mathcal{L}_{\Pi^2_C}} c : l_2\end{gathered}$$

  The result follows immediately from the induction hypothesis.

□

Lemma 10. *For every program c, security labelling $\Gamma : Var \to \mathcal{P}(P)$, and two integrity policies generated by the closure operators Π^1_I, Π^2_I respectively, such that $\Pi^1_I \sqsubseteq \Pi^2_I$, if there is a security level $l_1 \subseteq P$ such that:*

$$\Pi^1_I \circ \Gamma \vdash_{\mathcal{L}_{\Pi^1_I}} c : l_1$$

Then, there is a security level $l_2 \subseteq P$ such that:

$$l_1 \subseteq l_2 \;\wedge\; \Pi^2_I \circ \Gamma \vdash_{\mathcal{L}_{\Pi^2_I}} c : l_1$$

Proof. The proof proceeds by induction on the derivation $\Pi_I^1 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^1}} c : l_1$. Without loss of generality, the security levels corresponding to the writing effects of the typing judgements are considered to be closed set under the corresponding closure operator.

- **assign:** $\mathtt{x} := \mathtt{e}$. By hypothesis we know that $\Pi_I^1(\Gamma(e)) \subseteq \Pi_I^1(\Gamma(x))$. So, lemma 7 guarantees that: $\Pi_I^2(\Gamma(e)) \subseteq \Pi_I^2(\Gamma(x))$. By hypothesis: $\Pi_I^1(\Gamma(x)) \subseteq \Pi_I^2(\Gamma(x))$, thus: $l_1 \subseteq l_2$.

- **seq:** $\mathtt{c}_1;\mathtt{c}_2$. The hypothesis ensures that there are security levels l_1' and and l_1'' such that:
$$\Pi_I^1 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^1}} c_1 : l_1'$$
$$\Pi_I^1 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^1}} c_2 : l_1''$$
$$l_1 = l_1' \cap l_1''$$
The induction hypothesis guarantees the existence two security levels l_2' and l_2'' such that:
$$l_1' \subseteq l_2' \ \wedge\ \Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} c_1 : l_2'$$
$$l_1'' \subseteq l_2'' \ \wedge\ \Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} c_2 : l_2''$$
It follows that: $l_1' \cap l_1'' \subseteq l_2' \cap l_2''$.

- **if:** $\mathtt{if\ b\ then\ c}_1\ \mathtt{else\ c}_2$. The hypothesis ensures that there are security levels l_1' and and l_1'' such that:
$$\Pi_I^1 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^1}} c_1 : l_1'$$
$$\Pi_I^1 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^1}} c_2 : l_1''$$
$$\Pi_I^1(\textstyle\bigcup_{v \in vars(b)} \Gamma(v)) \subseteq l_1' \cap l_1''$$
The induction hypothesis guarantees the existence two security levels l_2' and l_2'' such that:
$$l_1' \subseteq l_2' \ \wedge\ \Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} c_1 : l_2'$$
$$l_1'' \subseteq l_2'' \ \wedge\ \Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} c_2 : l_2''$$
It follows:
$$\begin{array}{l} l_1' \cap l_1'' \subseteq l_2' \cap l_2'' \\ \Pi_I^1(\Gamma(b)) \subseteq l_1' \cap l_1'' = \Pi_I^1(l_1' \cap l_1'') \quad \text{(hypothesis)} \\ \Pi_I^1(\Gamma(b)) \subseteq \Pi_I^1(l_2' \cap l_2'') \quad \text{(monotonocity and transitivity)} \\ \Pi_I^2(\Gamma(b)) \subseteq \Pi_I^2(l_2' \cap l_2'') = l_2' \cap l_2'' \quad \text{(lemma 7)} \end{array}$$

- **while:** $\mathtt{while\ b\ do\ c}$. The hypothesis ensures that there is a security level l_1 such that:
$$\Pi_I^1 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^1}} c : l_1$$
$$\Pi_I^1(\Gamma(b)) \subseteq l_1$$

The induction hypothesis guarantees the existence of a security level l_2 such that:

$$\Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} c : l_2$$
$$l_1 \subseteq l_2$$

Thus applying lemma 7, it follows that:

$$\Pi_I^1(l_1) \subseteq \Pi_I^1(l_2)$$
$$\Pi_I^1(\Gamma(b)) \subseteq \Pi_I^1(l_2)$$
$$\Pi_I^2(\Gamma(b)) \subseteq \Pi_I^2(l_2) = l_2$$

□

4 Computing the strictest integrity policy

This section addresses the problem of computing the strictest integrity policy to which a program complies. It presents a static analysis that, given a program c and an initial security labelling $\Gamma : Var \to \mathcal{P}(P)$ computes the least (and thus the strictest) closure operator Π_I such that:

$$\Pi_I \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I}} c : l_I \tag{13}$$

for some security level $l_I \subseteq P$. As discussed in section 2, establishing a closure operator over the original lattice requires reinterpreting the security labelling.

In this section, security lattices generated by additive closure operators are considered separately, since they can be represented by flow functions. As such, this section is divided in two subsections. The first presents an analysis which considers all security lattices generated by arbitrary closure operators. The second argues that it is not possible to infer the strictest additive closure operator to which a program complies since it may not exist.

4.1 Inferring the strictest integrity policy generated by a closure operator

We shall now introduce a static analysis that given a program c, a security labelling Γ, the security level of the context under which the program is being typed l and a given closure operator Π_I^0, returns a new closure operator Π_I^1 such that c is typable under the security lattice generated by Π_I^1 and the security labelling $\Pi_I^1 \circ \Gamma$ in a security context of level $\Pi_I^1(l)$.

Definition 8 (Strictest integrity policy generated by an arbitrary closure operator)**.**

SKIP $\mathcal{F}_\Gamma[[\texttt{skip}]](l,\Pi_I^0) = \Pi_I^0$

ASSIGN
$$\mathcal{F}_\Gamma[[\texttt{x} := \texttt{e}]](l,\Pi_I^0) = \Pi_I^1$$
$$\Pi_I^1(J) = \begin{cases} \Pi_I^0(J) & \text{if } \Pi_I^0(\Gamma(x)) \notin \Pi_I^0(J) \\ \Pi_I^0(\Gamma(e) \cup l \cup J) & \text{if } \Pi_0^1(\Gamma(x)) \subseteq \Pi_I^0(J) \end{cases}$$

SEQ $\mathcal{F}_\Gamma[[\texttt{c}_1;\texttt{c}_2]](l,\Pi_I^0) = \mathcal{F}_\Gamma[[c_2]](l,\mathcal{F}_\Gamma[[c_1]](l,\Pi_I^0))$

IF $\mathcal{F}_\Gamma[[\texttt{if b then c}_1\ \texttt{else c}_2]](l,\Pi_I^0) = \mathcal{F}_\Gamma[[c_2]](\Pi_I^0(\Gamma(b)\cup l),\mathcal{F}_\Gamma[[c_1]](\Pi_I^0(\Gamma(b)\cup l),\Pi_I^0))$

WHILE
$$\mathcal{F}_\Gamma[[\texttt{while b do c}]](l,\Pi_I^0) = lfp(H)$$
Where:
$$H(\Pi) = \mathcal{F}_\Gamma(\Pi(\Gamma(b)\cup l),\Pi) \sqcup \Pi_I^0$$

Since this analysis (definition 8) aims to infer the smallest closure operator that generates an integrity policy which the program being analysed is compliant with, the output of the analysis must be a closure operator. Therefore, Lemma 11 provides the necessary motivation for the [assign] rule.

Lemma 11. *Given a closure operator* Π *over* $\mathcal{P}(P)$ *and two sets of principals* $X, Y \subseteq P$*, the smallest closure operator* Π' *such that* $\Pi \sqsubseteq \Pi'$ *and* $\Pi(X \cup Y) \subseteq \Pi'(X)$ *is given by:*

$$\Pi'(J) = \begin{cases} \Pi(J) & \text{if } \Pi(X) \notin \Pi(J) \\ \Pi(J \cup Y) & \text{if } \Pi(X) \subseteq \Pi(J) \end{cases}$$

Proof. The proof must be done in two steps: in the first step we shall prove that Π' is a closure operator, in the second steps we prove that it is indeed the samllest.

- Extensivity. For every set of principals J, $J \subseteq \Pi'(J)$. This follows from the fact that Π is a closure operator.
- Monotonocity. Consider two sets of principals $P_1, P_2 \subseteq P$, such that $P_1 \subseteq P_2$. If $\Pi(X) \notin \Pi(P_1)$, then:

  $\Pi'(P_1) = \Pi(P_1)$
  $\Pi(P_2) \subseteq \Pi'(P_2)$
  $\Pi'(P_1) \subseteq \Pi(P_2) \subseteq \Pi'(P_2)$ (applying transitivity and the monotonocity of Π)

  If $\Pi(X) \subseteq \Pi(P_1)$, then it follows by transitivity that: $\Pi(X) \subseteq \Pi(P_2)$. So:

  $$\Pi'(P_1) = \Pi(P_1 \cup Y)$$
  $$\Pi'(P_2) = \Pi(P_2 \cup Y)$$

  Applying the monotonocity of Π: $\Pi(P_1 \cup Y) \subseteq \Pi(P_2 \cup Y)$.

- Idempotence. For any given set of principals $J \subseteq P$: $\Pi'(\Pi'(J)) = \Pi'(J)$. Suppose $\Pi(X) \nsubseteq \Pi(J)$:

$$\Pi'(J) = \Pi(J)$$
$$\Pi(X) \nsubseteq \Pi'(J)$$
$$\Pi'(\Pi'(J)) = \Pi'(J) = \Pi(J)$$

  Suppose $\Pi(X) \subseteq \Pi(J)$:

$$\Pi'(J) = \Pi(J \cup Y)$$
$$\Pi(J) \subseteq \Pi'(J)$$
$$\Pi(X) \subseteq \Pi'(J)$$
$$\Pi'(\Pi'(J)) = \Pi'(J \cup Y)$$
$$\Pi(X) \subseteq \Pi(J \cup Y)$$
$$\Pi'(\Pi'(J)) = \Pi'(J \cup Y) = \Pi(J \cup Y \cup Y) = \Pi(J \cup Y) = \Pi'(J)$$

Before proceeding to second part of the proof, we will argue that if $Y \subseteq \Pi(X)$, then $\Pi' = \Pi$. That is, for every set of principals J: $\Pi'(J) = \Pi(J)$. If $\Pi(X) \subseteq \Pi(J)$, then:

$$\Pi'(J) = \Pi(J \cup Y)$$
$$Y \subseteq \Pi(X) \subseteq \Pi(J)$$
$$J \subseteq \Pi(J)$$
$$J \cup Y \subseteq \Pi(J)$$
$$\Pi(J \cup Y) \subseteq \Pi(\Pi(J)) = \Pi(J) \quad \text{(by monotonocity and idempotence of } \Pi)$$

It only remains to prove that Π' is the smallest closure operator which verifies: $\Pi \sqsubseteq \Pi'$ and $\Pi(X \cup Y) \subseteq \Pi'(X)$. So, we shall assume that there is a closure operator Π'' and a set of principals J such that:

$$\Pi \sqsubseteq \Pi''$$
$$\Pi(X \cup Y) \subseteq \Pi''(X)$$
$$\Pi''(J) \subset \Pi'(J)$$

Clearly, it must be the case that $Y \nsubseteq \Pi(X)$, otherwise $\Pi' = \Pi$ which contradicts the assumption. Also, $\Pi(X) \subseteq \Pi(J)$, otherwise $\Pi'(J) = \Pi(J)$ which contradicts the hypothesis. So:

$$\Pi(X) \subseteq \Pi(J) \Rightarrow \Pi''(X) \subseteq \Pi''(J) \quad \text{(Applying lemma 7)}$$
$$\Pi''(J) \subset \Pi'(J) \Leftrightarrow \Pi''(J) \subset \Pi(J \cup Y)$$
$$\Pi(X \cup Y) \cup \Pi''(X) \Rightarrow Y \subseteq \Pi''(X) \Rightarrow Y \subseteq \Pi''(J)$$
$$J \subseteq \Pi''(J)$$
$$J \cup Y \subseteq \Pi''(J) \Rightarrow \Pi''(J \cup Y) \subseteq \Pi''(J) \Rightarrow \Pi(J \cup Y) \subseteq \Pi''(J) \quad \text{(Contradiction)}$$

□

Lemma 12 guarantees that the analysis presented in Definition 8 is well-defined (in the sense that each input yiels a unique output), whereas Lemma 13 states that the output of the analysis is always a closure operator.

Lemma 12 (Existence and Monotonocity of the Analysis)**.** *For any program* `c`*, security labelling* $\Gamma : \mathit{Var} \to \mathcal{P}(P)$ *the following holds:*

- *For every security level* $l \subseteq P$ *and closure operator* Π_I^0*,* $\mathcal{F}_\Gamma[[c]](l, \Pi_I^0)$ *is well defined.*
- $\mathcal{F}_\Gamma[[\texttt{c}]]$ *is monotone.*

Proof. We proceed by induction on the structure of `c`.

- **skip:** `skip`.
  $\mathcal{F}_\Gamma[[\texttt{skip}]](l, \Pi_I^0) = \Pi_I^0$, so both claims hold automatically.
- **assign:** `x := e`.
  $\mathcal{F}_\Gamma[[\texttt{x := e}]](l, \Pi_I^0)$ is clearly uniquely defined.

  Suppose that $l_1 \subseteq l_2$ and $\Pi_I^1 \sqsubseteq \Pi_I^2$, we must show that for every security level $J \subseteq P$:

  $$\mathcal{F}_\Gamma[[\texttt{x:=e}]](l_1, \Pi_I^1)(J) \subseteq \mathcal{F}_\Gamma[[\texttt{x:=e}]](l_2, \Pi_I^2)(J).$$

  If $\Pi_I^1(\Gamma(x)) \not\subseteq \Pi_I^1(J)$, then the result follows immediately.
  If $\Pi_I^1(\Gamma(x)) \subseteq \Pi_I^1(J)$, then lemma 7 guarantees that $\Pi_I^2(\Gamma(x)) \subseteq \Pi_I^2(J)$ and the result also follows.
- **seq:** $\texttt{c}_1\texttt{;c}_2$.
  The uniqueness of $\mathcal{F}_\Gamma[[\texttt{c}_2]](l, \mathcal{F}_\Gamma[[\texttt{c}_1]](l, \Pi_I^0))$ follows directly from the induction hypothesis.

  Suppose that $l_1 \subseteq l_2$ and $\Pi_I^1 \sqsubseteq \Pi_I^2$, then applying the induction hypothesis:

  $$\mathcal{F}_\Gamma[[\texttt{c}_1]](l_1, \Pi_I^1) \sqsubseteq \mathcal{F}_\Gamma[[\texttt{c}_1]](l_2, \Pi_I^2)$$

  Applying the induction hypothesis yet again:

  $$\mathcal{F}_\Gamma[[\texttt{c}_2]](l_1, \mathcal{F}_\Gamma[[\texttt{c}_1]](l_1, \Pi_I^1)) \sqsubseteq \mathcal{F}_\Gamma[[\texttt{c}_2]](l_2, \mathcal{F}_\Gamma[[\texttt{c}_1]](l_2, \Pi_I^2))$$

- **if:** `if b then` $\texttt{c}_1$ `else` $\texttt{c}_2$.
  Applying the induction hypothesis one can easily check that $\mathcal{F}_\Gamma[[\texttt{if b then c}_1\texttt{ else c}_2]](l, \Pi_I^0)$ is uniquely defined.

  Suppose that $l_1 \subseteq l_2$ and $\Pi_I^1 \sqsubseteq \Pi_I^2$, then:

  $$\begin{array}{ll}
  \Gamma(b) \cup l_1 \subseteq \Gamma(b) \cup l_2 & \text{since: } l_1 \subseteq l_2 \\
  \Pi_I^1(\Gamma(b) \cup l_1) \subseteq \Pi_I^2(\Gamma(b) \cup l_1) & \text{since: } \Pi_I^1 \sqsubseteq \Pi_I^2 \\
  \Pi_I^2(\Gamma(b) \cup l_1) \subseteq \Pi_I^2(\Gamma(b) \cup l_2) & \\
  \Pi_I^1(\Gamma(b) \cup l_1) \subseteq \Pi_I^2(\Gamma(b) \cup l_2) & \text{by transitivity}
  \end{array}$$

Establishing:

$$l_1' = \Pi_I^1(\Gamma(b) \cup l_1)$$
$$l_2' = \Pi_I^2(\Gamma(b) \cup l_2)$$
$$l_1' \subseteq l_2'$$

Applying the induction hypothesis:

$$\mathcal{F}_\Gamma[[\mathtt{c}_1]](l_1', \Pi_I^1) \sqsubseteq \mathcal{F}_\Gamma[[\mathtt{c}_1]](l_2', \Pi_I^2)$$
$$\mathcal{F}_\Gamma[[\mathtt{c}_2]](l_1', \mathcal{F}_\Gamma[[\mathtt{c}_1]](l_1', \Pi_I^1)) \sqsubseteq \mathcal{F}_\Gamma[[\mathtt{c}_2]](l_2', \mathcal{F}_\Gamma[[\mathtt{c}_1]](l_2', \Pi_I^2))$$

- **while:** `while b do c`.
  The induction hypothesis guarantees that $\mathcal{F}_\Gamma[[c]]$ exists and is monotone. As such, the existence of the least fixed point follows from Kleen's theorem.

  It remains to prove the monotonocity claim. Thus, consider two security levels $l_1, l_2 \subseteq P$ and two flow functions Π_I^1 and Π_I^2 such that: $l_1 \subseteq l_2$ and $\Pi_I^1 \sqsubseteq \Pi_I^2$, which give rise to two distinct operators on flow operators.

  $$H_1(\Pi) = \mathcal{F}_\Gamma[[\mathtt{c}]](\Pi(\Gamma(b) \cup l_1), \Pi) \sqcup \Pi_I^1$$
  $$H_2(\Pi) = \mathcal{F}_\Gamma[[\mathtt{c}]](\Pi(\Gamma(b) \cup l_2), \Pi) \sqcup \Pi_I^2$$

  We shall now prove the following statement:

  $$\forall_{k \in N}. H_1^k(\bot) \sqsubseteq \mathrm{lfp}(H_2)$$

  by induction on k.

  - $k = 0$. $\bot \sqsubseteq \mathrm{lfp}(H_2)$.
  - Applying the induction hypothesis: $H_1^k(\bot) \sqsubseteq \mathrm{lfp}(H_2)$.
    $H_1^{k+1}(\bot) = H_1(H_1^k(\bot)) = \mathcal{F}_\Gamma[[\mathtt{c}]](H_1^k(\bot)(\Gamma(b) \cup l, H_1^k(\bot)) \sqcup \Pi_I^1$
    The outer induction hypothesis guarantees that $\mathcal{F}_\Gamma[[c]]$ is monotone. So:

    $$\mathcal{F}_\Gamma[[\mathtt{c}]](H_1^k(\bot)(\Gamma(b) \cup l_1), H_1^k(\bot)) \sqcup \Pi_I^1 \sqsubseteq \mathcal{F}_\Gamma[[\mathtt{c}]](\mathrm{lfp}(H_2)(\Gamma(b) \cup l_2), \mathrm{lfp}(H_2)) \sqcup \Pi_I^2$$

    Thus:
    $H_1^{k+1}(\bot) \sqsubseteq H_2(\mathrm{lfp}(H_2)) = \mathrm{lfp}(H_2)$

  The result now follows from the fact that the lattice of closure operators over a given finite set of principals is also finite.

□

Lemma 13. *For any program* $\mathtt{c}$*, security labelling* $\Gamma : Var \to \mathcal{P}(P)$*, security level* $l \subseteq P$ *and closure operator* Π_I^0*, such that:*

$$\Pi_I^1 = \mathcal{F}_\Gamma[[\mathtt{c}]](l, \Pi_I^0)$$

Then: Π_I^1 *is a closure operator.*

Proof. The proof proceeds by induction on the structure of c.

- **skip:** `skip`.
  $\mathcal{F}_\Gamma[[\texttt{skip}]](l, \Pi_I^0) = \Pi_I^0$. Thus the claim follows immediately.

- **assign:** `x := e`.
  $\Pi_I^1 = \mathcal{F}_\Gamma[[\texttt{x} := \texttt{e}]](l, \Pi_I^0)$
  Assuming that Π_0^I is a closure operator:

  - **Extensiveness:** It suffices to remark that:

    $$J \subseteq \Pi_I^0(J)$$
    $$J \subseteq \Pi_I^0(\Gamma(e) \cup l \cup J).$$

    Thus $J \subseteq \Pi_I^1(J)$.

  - **Monotonocity:** Consider two security levels $J_1, J_2 \subseteq P$, such that $J_1 \subseteq J_2$. If $\Pi_I^0(\Gamma(x)) \nsubseteq \Pi_I^0(J_1)$, the result immediately follows. If $\Pi_I^0(\Gamma(x)) \subseteq \Pi_I^0(J_1)$, then $\Pi_I^0(\Gamma(x)) \subseteq \Pi_I^0(J_2)$ (since $\Pi_I^0(J_1) \subseteq \Pi_I^0(J_2)$), which implies that $\Pi_I^1(J_1) \subseteq \Pi_I^1(J_2)$.

  - **Idempotence:** One must prove that for any $J \subseteq P$, $\Pi_I^1(\Pi_I^1(J)) \subseteq \Pi_I^1(J)$. The converse side follows from monotonocity and extensivity.
    Suppose $\Pi_I^0(\Gamma(x)) \nsubseteq \Pi_I^0(J)$. Then:

    $$\Pi_I^1(J) = \Pi_I^0(J)$$
    $$\Pi_I^0(\Gamma(x)) \nsubseteq \Pi_I^0(J) = \Pi_I^0(\Pi_I^0(J))$$
    $$\Pi_I^1(\Pi_I^1(J)) = \Pi_I^1(\Pi_I^0(J)) = \Pi_I^0(\Pi_I^0(J)) = \Pi_I^0(J)$$

    Suppose $\Pi_I^0(\Gamma(x)) \subseteq \Pi_I^0(J)$:

    $$\Pi_I^1(J) = \Pi_I^0(\Gamma(e) \cup l \cup J)$$
    $$J \subseteq J \cup l \cup \Gamma(e)$$
    $$\Rightarrow \Pi_I^0(J) \subseteq \Pi_I^0(J \cup l \cup \Gamma(e))$$
    $$\Rightarrow \Pi_I^0(J) \subseteq \Pi_I^0(\Pi_I^0(J \cup l \cup \Gamma(e)))$$
    $$\Pi_I^1(\Pi_I^1(J)) = \Pi_I^1(\Pi_I^0(\Gamma(e) \cup l \cup J))$$
    $$\Pi_I^1(\Pi_I^1(J)) = \Pi_I^0(\Gamma(e) \cup l \cup J)$$

- **seq:** $\texttt{c}_1\texttt{;c}_2$.
  Let us assume that Π_I^0 is a closure operator. Applying the induction hypothesis we get that:

  $$\Pi_I^1 = \mathcal{F}_\Gamma[[\texttt{c}_1]](l, \Pi_I^0)$$

  is a closure operator. So one may apply the induction hypothesis yet again and thus conclude that:

  $$\Pi_I^2 = \mathcal{F}_\Gamma[[\texttt{c}_2]](l, \Pi_I^1)$$

  is also a closure operator.

- **if:** `if b then` $\texttt{c}_1$ `else` $\texttt{c}_2$.
  Assume that Π_I^0 is a closure operator and that $l_b = \Pi_I^0(\Gamma(b) \cup l)$. Thus, applying the induction hypothesis:

- $\mathcal{F}_\Gamma[[\mathtt{c}_1]](l_b, \Pi_I^0)$ is a closure operator.
- $\mathcal{F}_\Gamma[[\mathtt{c}_1]](l_b, \mathcal{F}_\Gamma[[\mathtt{c}_2]](l_b, \Pi_I^0))$ is a closure operator.

- **while:** `while b do c`.
  The induction hypothesis guarantees that for any given security level $l \subseteq P$ and closure opertor Π_I: $\mathcal{F}_\Gamma[[\mathtt{c}]](l, \Pi_I)$ is a closure operator.
  Since:
  $$\mathcal{F}_\Gamma[[\texttt{while b do c}]](l, \Pi_I) = \bigsqcup_{k \in \mathcal{N}} \{H^k(\bot)\}$$
  $$H(\Pi) = \mathcal{F}_\Gamma[[\mathtt{c}]](\Pi(l \cup \Gamma(b)), \Pi) \sqcup \Pi_I$$
  We now proceed by induction on k.
  - $k = 0$. $\bot$ is a closure operator since it assigns to every security level the empty set.
  - The induction hypothesis guarantees that: $H^k(\bot)$ is a closure operator. Applying the exterior induction hypothesis, one gets that $H^{k+1}(\bot)$ is also a closure operator.

  The result now follows from the fact that the lattice of closure operators over a given finite set of principals is also finite.

□

Lemmas 11 and 14 clarify how the analysis works: it systematically raises the closure operator given as input in order to find the smallest closure operator that types the instruction being analysed.

Lemma 14 (Extensivity of the analysis)**.** *Given a program* $\mathtt{c}$*, a security labelling* $\Gamma : \mathit{Var} \to \mathcal{P}(P)$*, a security level* $l \subseteq P$*, a closure operator* Π_I^0 *on* $\mathcal{P}(P)$ *and an arbitrary set of principals* J*, then:*

$$\Pi_I^0(J) \subseteq \Pi_I^1(J)$$

where $\Pi_I^1(J) = \mathcal{F}_\Gamma[[c]](l, \Pi_I^0)$.

Proof. The proof proceeds by induction on the structure of c.

- **skip:** `skip`.
  $\Pi_I^0(J) \subseteq \Pi_I^0(J) = \Pi_I^1(J)$.

- **assign:** `x := e`.
  Observing that:
  $$\Pi_I^0(J) \subseteq \Pi_I^0(J)$$
  $$\Pi_I^0(J) \subseteq \Pi_I^0(J \cup \Gamma(e) \cup l)$$
  one can conclude: $\Pi_I^0(J) \subseteq \Pi_I^1(J)$.

- **seq:** $\mathtt{c}_1;\mathtt{c}_2$.
  Applying the induction hypothesis: $\Pi_I^0(J) \subseteq \mathcal{F}_\Gamma[[\mathtt{c}_1]](l, \Pi_I^0)(J)$.
  Applying the induction hypothesis again: $\mathcal{F}_\Gamma[[\mathtt{c}_1]](l, \Pi_I^0)(J) \subseteq \mathcal{F}_\Gamma[[\mathtt{c}_2]](l, \mathcal{F}_\Gamma[[\mathtt{c}_2]](l, \Pi_I^0))(J)$
  The result now follows from transitivity.

- **if:** `if b then c1 else c2`.
  The result follows in the same way as in the case of [seq].

- **while:** `while b do c`.
  $\Pi_I^1 = \bigsqcup_{k\in\mathcal{N}}\{H^k(\bot)\} \sqcup \Pi_I^0$. We will now show by induction on k that for any set of principals $J' \subseteq P$, $\Pi_I^0(J') \subseteq H^k(\bot)(J')$.
  - $k = 0$. $H^0(\bot) = \Pi_I^0$.
  - The induction hypothesis ensures that: $\Pi_I^0(J') \subseteq H^k(\bot)(J')$.
    $H^{k+1}(\bot)(J') = \mathcal{F}_\Gamma[[c]](l, H^k(\bot))(J')$.
    The exterior induction hypothesis implies that:
    $H^k(\bot))(J') \subseteq H^{k+1}(\bot)(J')$. The result now follows by transitivity.

□

Lemma 15 guarantees that the output of the analysis allows typing the program being analysed, whereas Theorem 1 guarantees that it is in fact the lowest closure operator that allows typing the program.

Lemma 15 (Soundness)**.** *Given a program* `c`*, a security labelling* $\Gamma : Var \to \mathcal{P}(P)$*, security level* $l \subseteq P$ *and closure operator* $\Pi_I^0 : \mathcal{P}(P) \to \mathcal{P}(P)$ *such that:*

$$\mathcal{F}_\Gamma[[\texttt{c}]](l_0, \Pi_I^0) = \Pi_I^1$$

Then there is a security level $l_1 \subseteq P$ *such that* $\Pi_I^0(l_0) \subseteq l_1$ *and* $\Gamma \circ \Pi_I^1 \vdash_{\mathcal{L}_{\Pi_I^1}} c : l_1$.

Proof. Without loss of generality, the writing effects of the typing judgements are always considered to be closed sets under the closure operator of the corresponding security lattice. The proof proceeds by induction on the structure of c.

- **skip:** `skip`.
  $\mathcal{F}_\Gamma[[\texttt{skip}]](l_0, \Pi_I^0) = \Pi_I^0$.
  $\Gamma \circ \Pi_I^0 \vdash_{\mathcal{L}_{\Pi_I^0}} \texttt{skip} : \Pi_I^0(P)$.
  $l_0 \subseteq P \Rightarrow \Pi_I^0(l_0) \subseteq \Pi_I^0(P) = P$

- **assign:** `x := e`.
  $\mathcal{F}_\Gamma[[\texttt{x} := \texttt{e}]](l_0, \Pi_I^0) = \Pi_I^1$.
  $\Pi_I^1(\Gamma(e)) \subseteq \Pi_I^0(\Gamma(e) \cup l) \subseteq \Pi_I^0(\Gamma(x) \cup l \cup \Gamma(e)) = \Pi_I^1(\Gamma(x))$
  As such: $\Gamma \circ \Pi_I^1 \vdash_{\mathcal{L}_{\Pi_I^1}} \texttt{x} := \texttt{e} : \Pi_I^1(\Gamma(x))$.
  Naturally: $\Pi_I^0(l) \subseteq \Pi_I^0(l \cup \Gamma(e) \cup \Gamma(x)) = \Pi_I^1(\Gamma(x))$.

- **seq:** `c1;c2`.
  $\mathcal{F}_\Gamma[[\texttt{c}_1;\texttt{c}_2]](l_0, \Pi_I^0) = \mathcal{F}_\Gamma[[\texttt{c}_2]](l_0, \mathcal{F}_\Gamma[[\texttt{c}_1]](l_0, \Pi_I^0))$. Applying the induction hypothesis:
  $$\Pi_I^1 = \mathcal{F}_\Gamma[[\texttt{c}_1]](l_0, \Pi_I^0)$$
  $$\Gamma \circ \Pi_I^1 \vdash_{\mathcal{L}_{\Pi_I^1}} c_1 : l_1$$
  $$\Pi_I^0(l_0) \subseteq l_1$$

Applyng the induction hypothesis again:

$$\begin{aligned}&\Pi_I^2 = \mathcal{F}_\Gamma[[\mathtt{c}_2]](l_0, \Pi_I^1)\\&\Gamma \circ \Pi_I^2 \vdash_{\mathcal{L}_{\Pi_I^2}} c_2 : l_2'\\&\Pi_I^0(l_0) \subseteq l_2'\end{aligned}$$

Lemma 14 guarantees that $\Pi_I^1 \sqsubseteq \Pi_I^2$. As such, applying lemma 10, one may conclude that:

$$\begin{aligned}&\Gamma \circ \Pi_I^2 \vdash_{\mathcal{L}_{\Pi_I^2}} c_1 : l_2''\\&l_1 \subseteq l_2''\end{aligned}$$

It follows that:

$$\begin{aligned}&\Gamma \circ \Pi_I^2 \vdash \mathcal{L}_{\Pi_I^2} \mathtt{c}_1; \mathtt{c}_2 : l_2' \cap l_2''\\&\Pi_I^0(l_0) \subseteq l_1 \subseteq l_2''\\&\Pi_I^0(l_0) \subseteq l_2'\\&\Pi_I^0(l_0) \subseteq l_2' \cap l_2''\end{aligned}$$

- **if:** `if b then c1 else c2`.
  $\mathcal{F}_\Gamma[[\mathtt{if\ b\ then\ c_1\ else\ c_2}]](l_0, \Pi_I^0) = \mathcal{F}_\Gamma[[\mathtt{c}_2]](l_b, \mathcal{F}_\Gamma[[\mathtt{c}_1]](l_b, \Pi_I^0))$.
  $l_b = \Pi_I^0(\Gamma(b) \cup l_0)$
  Applying the induction hypothesis:

$$\begin{aligned}&\Pi_I^1 = \mathcal{F}_\Gamma[[\mathtt{c}_1]](\Pi_I^0(\Gamma(b) \cup l_0), \Pi_I^0)\\&\Gamma \circ \Pi_I^1 \vdash_{\mathcal{L}_{\Pi_I^1}} c_1 : l_1\\&\Pi_I^0(\Gamma(b) \cup l_0) \subseteq l_1\end{aligned}$$

  Applying the induction hypothesis yet again:

$$\begin{aligned}&\Pi_I^2 = \mathcal{F}_\Gamma[[\mathtt{c}_2]](\Pi_I^0(\Gamma(b) \cup l_0)), \Pi_I^1)\\&\Gamma \circ \Pi_I^2 \vdash_{\mathcal{L}_{\Pi_I^2}} c_2 : l_2'\\&\Pi_I^1(\Pi_I^0(\Gamma(b) \cup l_0)) \subseteq l_2'\end{aligned}$$

  Lemma 14 guarantees that $\Pi_I^1 \sqsubseteq \Pi_I^2$. As such, applying lemma 10, one may conclude that:

$$\begin{aligned}&\Gamma \circ \Pi_I^2 \vdash_{\mathcal{L}_{\Pi_I^2}} c_1 : l_2''\\&l_1 \subseteq l_2''\end{aligned}$$

  It follows that:

$$\begin{aligned}&\Gamma \circ \Pi_I^2 \vdash_{\mathcal{L}_{\Pi_I^2}} \mathtt{if\ b\ then\ c_1\ else\ c_2} : l_2' \cap l_2''\\&\Pi_I^1(\Pi_I^0(\Gamma(b) \cup l_0)) \subseteq l_2' \Rightarrow \Pi_I^2(\Gamma(b)) \subseteq l_2'\\&\Pi_I^0(\Gamma(b) \cup l_0) \subseteq l_1 \subseteq l_2'' \Rightarrow \Pi_I^2(\Gamma(b)) \subseteq l_2''\\&\Pi_I^2(\Gamma(b)) \subseteq l_2' \cap l_2''\end{aligned}$$

- **while:** `while b do c`.
  $\Pi_I^1 = \mathcal{F}_\Gamma[[c]](\Pi_I^1(l_0 \cup \Gamma(b)), \Pi_I^1) \sqcup \Pi_I^0$.
  The induction hypothesis may be applied:

  $$\Pi_I^1 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^1}} c : l_1$$
  $$\Pi_I^1(\Pi_I^1(l_0 \cup \Gamma(b))) \subseteq l_1 \Rightarrow \Pi_I^1(l_0 \cup \Gamma(b)) \subseteq l_1$$

  Thus: $\Pi_I^1 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^1}} \texttt{while b do c} : l_1$.

□

Theorem 1 (Optimality of the analysis)**.** *Given a program* `c`*, a security labelling* Γ*, a security level* $l_1 \subseteq P$ *and closure operator* Π_I^0 *on* $\mathcal{P}(P)$ *such that:*

$$\mathcal{F}_\Gamma[[\texttt{c}]](l_1, \Pi_I^0) = \Pi_I^1$$

If there is a closure operator Π_I^2 *and a security level* $l_2 \subseteq P$ *such that:*

$$\Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} c : l_2$$
$$\Pi_I^0 \sqsubseteq \Pi_I^2 \quad l_1 \subseteq l_2$$

Then: $\Pi_I^1 \subseteq \Pi_I^2$.

Proof. The proof proceeds by induction on the structure of c.

- **skip:** `skip`.
  $\mathcal{F}_\Gamma[[\texttt{skip}]](l_1, \Pi_I^0) = \Pi_I^0 \sqsubseteq \Pi_I^2$

- **assign:** `x := e`.
  $\mathcal{F}_\Gamma[[\texttt{x := e}]](l_1, \Pi_I^0) = \Pi_I^1$.
  Lemma 11 guarantees that Π_I^1 is the smallest closure operator such that:

  $$\Pi_I^0 \sqsubseteq \Pi_I^1 \quad \Pi_I^0(\Gamma(x) \cup \Gamma(e) \cup l_1) \subseteq \Pi_I^1(\Gamma(x))$$

  Since $\Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} \texttt{x := e} : \Pi_I^2(\Gamma(x))$, it follows that:

  $$\Pi_I^2(\Gamma(e)) \subseteq \Pi_I^2(\Gamma(x)) \quad l_1 \subseteq \Pi_I^2(\Gamma(x))$$

  As such: $\Pi_I^2(\Gamma(x) \cup \Gamma(e) \cup l_1) \subseteq \Pi_I^2(\Gamma(x))$, which implies that:

  $$\Pi_I^0(\Gamma(x) \cup \Gamma(e) \cup l_1) \subseteq \Pi_I^2(\Gamma(x))$$

- **seq:** $\texttt{c}_1\texttt{;c}_2$.
  $\mathcal{F}_\Gamma[[\texttt{c}_1\texttt{;c}_2]](l_1, \Pi_I^0) = \mathcal{F}_\Gamma[[\texttt{c}_2]](l_1, \mathcal{F}_\Gamma[[\texttt{c}_1]](l_1, \Pi_I^0))$. Suppose that there is a closure operator Π_I^2 such that:

  $$\Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} \texttt{c}_1\texttt{;c}_2 : l_2 \quad \Pi_I^0 \sqsubseteq \Pi_I^2 \quad l_1 \subseteq l_2$$

Then, there must exist two sets of principals $l_2', l_2'' \subseteq P$ such that:

$$\Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} \mathtt{c}_1 : l_2' \quad \Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} \mathtt{c}_2 : l_2'' \quad l_2 = l_2' \cap l_2''$$

As such observing that:

$$\Pi_I^{11} = \mathcal{F}_\Gamma[[\mathtt{c}_1]](l_1, \Pi_I^0) \quad \Pi_I^0 \sqsubseteq \Pi_I^2 \quad \Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} \mathtt{c}_1 : l_2' \quad l_1 \subseteq l_2'$$

It is possible to apply the induction hypothesis and conclude that: $\Pi_I^{11} \sqsubseteq \Pi_I^2$. Again, observing that:

$$\Pi_I^{12} = \mathcal{F}_\Gamma[[\mathtt{c}_2]](l_1, \Pi_I^{11}) \quad \Pi_I^{11} \sqsubseteq \Pi_I^2 \quad \Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} \mathtt{c}_1 : l_2'' \quad l_1 \subseteq l_2''$$

It is possible to apply the induction hypothesis and conclude that: $\Pi_I^{12} \sqsubseteq \Pi_I^2$.

- **if:** `if b then c1 else c2`.
  Suppose that there is a closure operator Π_I^2 such that:

$$\Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} \mathtt{if\ b\ then\ c}_1\ \mathtt{else\ c}_2 : l_2 \quad \Pi_I^0 \sqsubseteq \Pi_I^2 \quad l_1 \subseteq l_2$$

  Then, there must exist two sets of principals $l_2', l_2'' \subseteq P$ such that:

$$\Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} \mathtt{c}_1 : l_2' \quad \Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} \mathtt{c}_2 : l_2'' \quad \Pi_I^2(\Gamma(b)) \subseteq l_2 \quad l_2 = l_2' \cap l_2''$$

  It is clear that: $\Pi_I^0(\Gamma(b) \cup l_1) \subseteq l_2' \cap l_2''$. Thus, observing that:

$$\Pi_I^{11} = \mathcal{F}_\Gamma[[\mathtt{c}_1]](\Pi_I^0(l_1 \cup \Gamma(b)), \Pi_I^0) \quad \Pi_I^0 \sqsubseteq \Pi_I^2 \quad \Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} \mathtt{c}_1 : l_2' \quad \Pi_I^0(l_1 \cup \Gamma(b)) \subseteq l_2'$$

  Applying the induction hypothesis, it follows that: $\Pi_I^{11} \sqsubseteq \Pi_I^2$. Again, observing that:

$$\Pi_I^{12} = \mathcal{F}_\Gamma[[\mathtt{c}_1]](\Pi_I^0(l_1 \cup \Gamma(b)), \Pi_I^{11}) \quad \Pi_I^{11} \sqsubseteq \Pi_I^2 \quad \Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} \mathtt{c}_2 : l_2'' \quad \Pi_I^0(l_1 \cup \Gamma(b)) \subseteq l_2''$$

  Applying the induction hypothesis, it follows that: $\Pi_I^{12} \sqsubseteq \Pi_I^2$.

- **while:** `while b do c`.
  Suppose that there is a closure operator Π_I^2 such that:

$$\Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} \mathtt{while\ b\ do\ c} : l_2 \quad \Pi_I^0 \sqsubseteq \Pi_I^2 \quad l_1 \subseteq l_2$$

  Then, it must be the case that:

$$\Pi_I^2 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^2}} \mathtt{c} : l_2 \quad \Pi_I^2(\Gamma(b)) \subseteq l_2$$

  So it follows that: $\Pi_I^0(\Gamma(b) \cup l_1) \subseteq l_2$. As such, one can apply the induction hypothesis (as in the previous cases) and conclude that: $\Pi_I^1 \sqsubseteq \Pi_I^2$.

□

Lemma 16. *Given a program* $\mathtt{c}$*, a security labelling* $\Gamma : Var \to \mathcal{P}(P)$*, a security level* $l \subseteq P$ *and a closure operator* Π_I^0 *on* $\mathcal{P}(P)$ *such that:*

$$\Pi_I^0 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^0}} c : l_0 \quad l_1 \subseteq l_0$$

Then: $\mathcal{F}_\Gamma[[\mathtt{c}]](l_1, \Pi_I^0) = \Pi_I^0$

Proof. Observing that:

$$\begin{array}{l} \Pi_I^1 = \mathcal{F}_\Gamma[[c]](l_1, \Pi_I^0) \\ \Pi_I^0 \circ \Gamma \vdash_{\mathcal{L}_{\Pi_I^0}} c : l_0 \\ l_1 \subseteq l_0 \quad \Pi_I^0 \sqsubseteq \Pi_I^0 \end{array}$$

It is possible to apply theorem 1 and conclude that: $\Pi_I^1 \sqsubseteq \Pi_I^0$. Applying lemma 14, it follows that: $\Pi_I^1 = \Pi_I^0$. □

Lemma 17 (Idempotence of the analysis)**.** *Given a program* $\mathtt{c}$*, a security labelling* $\Gamma : Var \to \mathcal{P}(P)$*, a security level* $l \subseteq P$ *and a closure operator* Π_I^0 *on* $\mathcal{P}(P)$*, the following holds:*

$$\mathcal{F}_\Gamma[[c]](l, \mathcal{F}_\Gamma[[c]](l, \Pi_I^0)) = \mathcal{F}_\Gamma[[c]](l, \Pi_I^0)$$

Proof. This lemma is an immediate consequence of lemmas 16 and 15. □

Lemma 18. *Given a program* $\mathtt{c}$*, a security labelling* $\Gamma : Var \to \mathcal{P}(P)$ *and a security level* $l \subseteq P$*, the operator on closure operators defined by:*

$$\begin{array}{l} \Psi : (\mathcal{P}(P) \to \mathcal{P}(P)) \to (\mathcal{P}(P) \to \mathcal{P}(P)) \\ \Psi(\Pi) = \mathcal{F}_\Gamma[[c]](l, \Pi) \end{array}$$

is a closure operator.

Proof. This lemma is an immediate consequence of lemmas 17, 14 and 12. □

4.2 Inferring the strictest integrity policy generated by a flow function

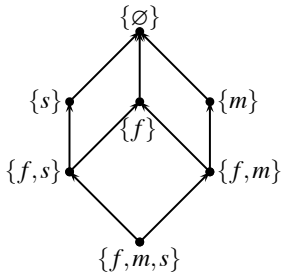
This section aims to clarify that in many situations there may not exist a *strictest* flow function (corresponding to an integrity policy) to which a program complies.

Section 4 clarifies how to compute the strictest closure operator (corresponding to an integrity policy) to which a given program complies. Having inferred it (Π_I), the problem of computing the strictest flow function corresponding to the same program consists in finding the smallest additive closure operator (Π_I') such that:

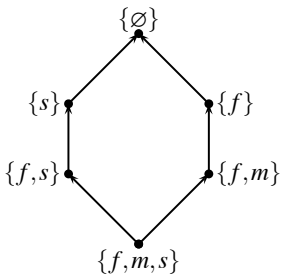
$$\Pi_I \sqsubseteq \Pi_I' \tag{14}$$

However, Π_I' may not exist, as shown in example 3.

Example 3. *Consider a family composed by three elements: father (f), mother (m) and son (s). The members of this family wish to implement an integrity policy such that the father is allowed to change all information that both mother and son can change. This integrity policy is expressed by the following lattice:*

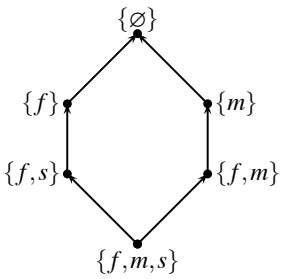


However there are two additive closure operators which are greater than the one which the family members wish to enforce but are not related to each other.



$f \to f \quad m \to f,m \quad s \to s$

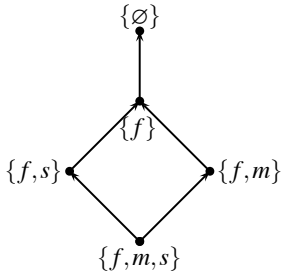
(1)



$f \to f \quad m \to m \quad s \to f,s$

(2)

Both the flow functions specified above generate closure operators which are less strict than the original one. However, they are not comparable. As such, this example illustrates a situation in which it is not possible to compute the strictest flow function which generates a closure operator that is less strict than the specified one.
Nevertheless, it is possible to compute an upper bound on the two flow functions specified above:



$$f \to f \quad m \to m, f \quad s \to s, f$$

Example 3 suggests computing an upper bound on the set flow functions which are generated by closure operators which are less strict than the specified one but not comparable with each other. This upper bound can be directly computed from the original closure operator as stated by lemma 19.

Lemma 19. *Given a closure operator* Π*, define the following flow function:*

$$\pi(p) = \Pi(\bigcup\{\Pi(X) \cap X^c | p \in X\} \cup p)$$

If there is a flow function π_1 *such that* $\Pi \sqsubseteq \pi_1^*$ *and* $\pi_1 \sqsubseteq \pi$*, then there is also a flow function* π_2 *such that* $\Pi \sqsubseteq \pi_2^*$*,* $\pi_2 \sqsubseteq \pi$*,* $\pi_1 \sqcup \pi_2 = \pi$ *and* $\pi_1 \not\sqsubseteq \pi_2$*.*

Proof. Suppose there is a flow function π_1 such that: $\Pi \sqsubseteq \pi_1^*$ and $\pi_1 \sqsubseteq \pi$. Then define π_2 in the following way:

$$\pi_2(p) = \Pi(\bigcup\{\Pi(X) \cap X^c \cap \pi_1(p) | p \in X\} \cup p)$$

□

5 Dependency analysis

Amtoft *et al* [4] introduce a Hoare Logic to reason about input output independences for simple `WHILE` programs. Ulteriorly, Hunt *et al* [13], study a family of flow-sensitive type systems for tracking information flow in `WHILE` programs. "This family is indexed by the choice of a flow lattice". They further notice that if the flow lattice is chosen to be the lattice $\mathcal{P}(Var)$, the corresponding flow-sensitive type system would match the De Morgan dual of Amtoft *et al* Hoare style independence logic.

This section instantiates the analysis presented in section 4 for the lattice $\mathcal{P}(Var)$. In a certain sense, this corresponds to inferring the strictest integrity policy to which a program complies, taking *Var* as the set of principals.

5.1 Semantic Characterization

This section clarifies the notion of dependency used in this report. Informally, a program is said to entail an independence between x and y (x does not depend on y), if for any memories that only differ on y, the sequence of values assigned to x is the same. Additionally, the definition of independence which we propose is termination insensitive since it only considers converging configurations.

Definition 9. *The* X*-projection of the trace* t*, written* $t|_X$*, where* $X \subseteq Var$ *and:*

$$t = \langle c_0, M_0 \rangle \overset{\alpha_0}{\to} \langle c_1, M_1 \rangle \overset{\alpha_1}{\to} \cdots \langle c_i, M_i \rangle \overset{\alpha_i}{\to} \cdots$$

is the sequence of memories:

$$m = M_0|_X, M_{i_1}|_X, M_{i_2}|_X, \cdots$$

Such that: $0 < i_1 < i_2 < \cdots$*; for every transition* $\langle c_j, M_j \rangle \xrightarrow{\alpha_j} \langle c_{j+1}, M_{j+1} \rangle$ *in t, if* $\alpha_j = x$ *for some* $x \in X$*, then* $j = i_k$ *for some k; and for every* i_k *in m there is a transition* $\langle c_j, M_j \rangle \xrightarrow{\alpha_j} \langle c_{j+1}, M_{j+1} \rangle$ *in t such that* $j = i_k$ *and* $\alpha_j = x$ *for some* $x \in X$*.*

The restriction of memory M to the variable set $X \subseteq Var$ is defined as the restriction of the original mapping corresponding to M to the variables in X. In order to simplify the notation, this report uses the abbreviation $M_1 =_X M_2$ to denote that:

$$\forall_{v \in X} \cdot M_1(v) = M_2(v) \tag{15}$$

Definition 10. *A program c is said to entail an independence between the sets of variables* $X, Y \subseteq Var$ *(X does not depend on Y), which is denoted by* $c \Vdash [X \propto Y]$*, if:*

$$\forall M_1, M_2 \cdot \begin{array}{c} M_1 =_{Var-Y} M_2 \\ \wedge \\ < c, M_1 > \Downarrow \wedge < c, M_2 > \Downarrow \end{array} \Rightarrow Trace(< c, M_1 >)|_X = Trace(< c, M_2 >)|_X$$

Closure operators and flow functions, which were introduced for confidentiality and integrity analysis may also be used for dependency analysis. In this case, flow functions are particularly useful, since they can be understood as *dependency functions*, that is, functions which map each variable to the set of variables on which it depends. Given a program c and a dependency function Δ, definition 11 specifies in which conditions c is said to entail Δ.

Definition 11. *A program c is said to entail a dependency function* Δ *(written* $c \Vdash \Delta$*) if:*

$$\forall x \in Var \ \forall Y \subseteq Var \cdot Y \subseteq Var - \Delta(x) \Rightarrow c \Vdash [x \propto Y]$$

Clearly, in dependency analysis the security labelling which maps each variable x to the corresponding sigleton set $\{x\}$ plays a central role. Thus for all $x \in Var$, $\Gamma_0(x) \stackrel{def}{=} \{x\}$.

Lemma 20. *A program c satisfies trace noninterference with respect to the integrity lattice generated by a given flow function* $\Delta : Var \rightarrow \mathcal{P}(Var)$ *and the security labelling* Γ_0 *(*$\Gamma_0 \Vdash_{\mathcal{L}_{\pi^*}} c$*) if and only, when taken as a dependency function, c entails* Δ *(*$c \Vdash \Delta$*).*

Proof. For the direct side of the equivalence, suppose that: $Y \subseteq Var - \Delta(x)$ for an arbitrary variable x and an arbitrary set of variables Y. Considering two memories, M_1, M_2 such that $M_1 =_{Var-Y} M_2$:

$M_1 =_{Var-Y} M_2$
$M_1 =_{\Delta(x)} M_2$ (since $Y \subseteq Var - \Delta(x)$)
$Trace(< c, M_1 >)|^{\Gamma_0, \mathcal{L}_{\Delta^*}}_{\Delta(x)} = Trace(< c, M_2 >)|^{\Gamma_0, \mathcal{L}_{\Delta^*}}_{\Delta(x)}$ (applying the hypothesis)
$Trace(< c, M_1 >)|_x = Trace(< c, M_2 >)_x$

Both configurations are assumed to converge, otherwise they may not be considered. For the converse side of the equivalence, suppose that program c entails a given dependency function Δ, one has to prove that c verifies trace non-interference under the

integrity lattice generated by the closure operator Δ^* and the security labelling Γ_0. For a given set of variables $X \subseteq Var$ and two memories M_1, M_2 such that $M_1 =_{\Delta^*(X)} M_2$, $< c, M_1 >\Downarrow$ and $< c, M_2 >\Downarrow$:

$$\begin{array}{l}
\forall_{x \in \Delta^*(X)}. \Delta(x) \subseteq \Delta^*(X) \\
\forall_{x \in \Delta^*(X)}. Var - \Delta^*(X) \subseteq Var - \Delta(x) \\
\forall_{x \in \Delta^*(X)}. c \Vdash [x \propto Var - \Delta^*(X)] \\
\forall_{x \in \Delta^*(X)}. Trace < c, M_1 > |_x = Trace < c, M_2 > |_x \\
Trace < c, M_1 > |_{\Delta^*(X)} = Trace < c, M_2 > |_{\Delta^*(X)}
\end{array}$$

□

It is important to emphasize that lemma 11 requires Δ to be a flow function instead of a general closure operator over $\mathcal{P}(Var)$ for the equivalence to hold. However, if, instead of Δ, a general closure operator is considered, the direct side of the implication still holds. As such, the integrity calculus introduced in definition 8 can be used to infer the smallest set of trace dependencies entailed by a given program.

5.2 Integrity analysis as a calculus of dependencies

In spite of being possible to instantiate the integrity analysis presented in definition 8 directly as a calculus of dependencies, when the input of a dependency analysis is an additive closure operator over $\mathcal{P}(P)$, Lemma 21 establishes that the output is also an additive closure operator. Since the smallest closure operator that can be considered corresponds to the identity closure operator (which is additive), one can conclude that dependency analysis can be formulated directly on flow functions instead of arbirary closure operators. Therefore, the general integrity analysis presented in definition 8 can be simplified when instantiated for dependency analysis. Definition 12 presents the corresponding calculus of trace dependencies.

Lemma 21. *Given a program c, an additive closure opterator $\Delta_0 : \mathcal{P}(Var) \to \mathcal{P}(Var)$ and a set $X \subseteq Var$ such that:*

$$\mathcal{F}_{\Gamma_0}[[c]](X, \Delta_0) = \Delta$$

Then, Δ is also an additive closure operator.

Proof. Lemma 13 guarantees that Δ is a closure operator, as such for all $Y \subseteq Var$:

$$\bigcup_{y \in Y} \Delta(\{y\}) \subseteq \Delta(Y)$$

It remains to show the converse inclusion:

$$\Delta(Y) \subseteq \bigcup_{y \in Y} \Delta(\{y\})$$

The proof proceeds by induction on the structure of c.

- **skip:** `skip`.
  $\mathcal{F}_{\Gamma_0}[[c]](X, \Delta_0) = \Delta_0$
  Since Δ_0 is an additive closure operator, the result follows.

- **assign:** `x := e`.
  $\mathcal{F}_{\Gamma_0}[[\texttt{x := e}]](X, \Delta_0) = \Delta$.

$$\Delta(Y) = \begin{cases} \Delta_0(Y) & \text{if } \Delta_0(\{x\}) \nsubseteq \Delta_0(Y) \\ \Delta_0(Y \cup X \cup vars(e)) & \text{if } \Delta_0(\{x\}) \subseteq \Delta_0(Y) \end{cases}$$

  There are two cases two consider:

  – $\Delta_0(\{x\}) \nsubseteq \Delta_0(Y)$. It follows that:

$$\Delta(Y) = \Delta_0(Y) = \bigcup_{y \in Y} \Delta_0(y)$$

    Clearly, for all $y \in Y$, $\Delta_0(\{x\}) \nsubseteq \Delta_0(\{y\})$, thus:

$$\Delta(Y) = \bigcup_{y \in Y} \Delta_0(y) = \bigcup_{y \in Y} \Delta(y)$$

  – $\Delta_0(\{x\}) \subseteq \Delta_0(Y)$. There is a variable $y_k \in Y$ such that $\Delta_0(\{x\}) \subseteq \Delta(\{y_k\})$. Thus:

$$\Delta(\{y_k\}) = \Delta_0(\{y_k\} \cup X \cup vars(e))$$
$$\Delta(Y) = \Delta_0(Y \cup X \cup vars(e)) = \Delta_0(Y - \{y_k\}) \cup \Delta_0(\{y_k\} \cup X \cup vars(e))$$
$$\Delta(Y) = \Delta_0(Y - \{y_k\}) \cup \Delta(y_k)$$

    Observing that for all $v \in Var$ $\Gamma_0(\{v\}) \subseteq \Gamma(\{v\})$. It follows that:

$$\Delta(Y) = \Delta_0(Y - \{y_k\}) \cup \Delta(y_k) = \left(\bigcup_{y \in Y - \{y_k\}} \Delta_0(y) \right) \cup \Delta(y_k) \subseteq \bigcup_{y \in Y} \Delta(y)$$

- **seq:** $\texttt{c}_1\texttt{;c}_2$.
  $\mathcal{F}_{\Gamma_0}[[\texttt{c}_1\texttt{;c}_2]](X, \Delta_0) = \mathcal{F}_{\Gamma_0}[[\texttt{c}_2]](X, \mathcal{F}_{\Gamma_0}[[\texttt{c}_1]](X, \Delta_0))$.
  The result follows immediately from the induction hypothesis.

- **if:** `if b then` $\texttt{c}_1$ `else` $\texttt{c}_2$.
  $\mathcal{F}_{\Gamma_0}[[\texttt{if b then c}_1\texttt{ else c}_2]](X, \Delta_0) = \mathcal{F}_{\Gamma_0}[[\texttt{c}_2]](X_b), \mathcal{F}_{\Gamma}[[\texttt{c}_1]](\Delta_0(X_b), \Delta_0))$.
  $X_b = \Delta_0(X \cup vars(b))$
  The result follows immediately from the induction hypothesis.

- **while:** `while b do c`.

$$\mathcal{F}_{\Gamma_0}[[\texttt{while b do c}]](X, \Delta_0) = \bigsqcup_{k \in \mathcal{N}} \{H^k(\bot)\}$$
$$H(\Delta) = \mathcal{F}_{\Gamma_0}[[\texttt{c}]](\Delta(X \cup vars(b)), \Delta) \sqcup \Delta_0$$

  The proof proceeds by induction on k.

  – k=0. $H^0(\bot) = \bot$.

– k+1. $H^{k+1}(\bot) = H(H^k(\bot)) = \mathcal{F}_{\Gamma_0}[[\mathtt{c}]](\Delta(X \cup vars(b)), H^k(\bot)) \sqcup \Delta_0$.
The inner induction hypothesis ensures that $H^k(\bot)$ is additive. The outer induction hypothesis ensures that $\mathcal{F}_{\Gamma_0}[[\mathtt{c}]](\Delta(X \cup vars(b)), H^k(\bot))$ is additive. Thus, the result follows by noting that Δ_0 is also assumed to be additive.

□

Definition 12 (A calculus for trace dependencies)**.**

SKIP
$$\mathcal{D}[[\mathtt{skip}]](l, \Delta_0) = \Delta_0$$

ASSIGN
$$\mathcal{D}[[\mathtt{x := e}]](X, \Delta_0) = \Delta_1$$
$$\Delta_1(y) = \begin{cases} \Delta_0(y) & \text{if } x \notin \Delta_0(y) \\ \Delta_0(y) \cup \Delta_0^*(X \cup vars(e)) & \text{if } x \in \Delta_0(y) \end{cases}$$

SEQ
$$\mathcal{D}[[\mathtt{c}_1; \mathtt{c}_2]](X, \Delta_0) = \mathcal{D}[[c_2]](X, \mathcal{D}[[c_1]](X, \Delta_0))$$

IF
$$\mathcal{D}[[\mathtt{if\ b\ then\ c}_1\ \mathtt{else\ c}_2]](X, \Delta_0) = \mathcal{D}[[c_2]](X_b, \mathcal{D}[[c_1]](X_b, \Delta_0))$$
$$X_b = \Delta_0^*(vars(b) \cup X$$

WHILE
$$\mathcal{D}[[\mathtt{while\ b\ do\ c}]](X, \Delta_0) = lfp(H)$$
Where:
$$H(\Delta) = \mathcal{D}(\Delta^*(vars(b) \cup X, \Delta) \sqcup \Delta_0$$

5.3 Dependencies as Types

Section 5.2 argues that when instantiating integrity analysis for the lattice $\mathcal{P}(Var)$, if the closure operator corresponding to the dependencies given as input is additive then so is the closure operator corresponding to the output. As such, a simplified dependency calculus that deals with flow functions instead of closure operators is introduced in definition 12. This calculus of dependencies evinces the fact that a dependency function can be also interpreted as a security labelling rather than a security lattice. Lemma 22 formalizes this dual interpretation.

Lemma 22. *Given a program c, a flow function $\Delta : Var \to \mathcal{P}(Var)$ and a set of variables X, then the following holds:*

$$\Gamma_0 \vdash_{\mathcal{L}_{\Delta^*}} c : X \Leftrightarrow \Delta \vdash_{\mathcal{P}(Var)} c : X$$

Proof. We prove the direct implication, the converse one is obtained in the exact same way. The proof proceeds by induction on the struture of c.

- **skip:** `skip`.
  $\Gamma_0 \vdash_{\mathcal{L}_{\Delta^*}} \texttt{skip} : Var$
  $\Delta \vdash_{\mathcal{P}(Var)} c : Var$.

- **assign:** `x := e`.
  The hypothesis ensures that:

  $$\Delta^*(\bigsqcup_{v \in vars(e)} \Gamma_0(v)) \subseteq \Delta^*(\Gamma_0(x))$$
  $$\bigsqcup_{v \in vars(e)} \Delta(v) \subseteq \Delta(x)$$

  It follows that: $\Delta \vdash_{\mathcal{P}(Var)} c : \Delta(x)$.

- **seq:** $\texttt{c}_1\texttt{;c}_2$.
  The hypothesis ensures that there are two sets of variables $X_1, X_2 \subseteq Var$ such that:

  $$\Gamma_0 \vdash_{\mathcal{L}_{\Delta^*}} \texttt{c}_1 : X_1$$
  $$\Gamma_0 \vdash_{\mathcal{L}_{\Delta^*}} \texttt{c}_2 : X_2$$

  Applying the induction hypothesis:

  $$\Delta \vdash_{\mathcal{P}(Var)} \texttt{c}_1 : X_1$$
  $$\Delta \vdash_{\mathcal{P}(Var)} \texttt{c}_2 : X_2$$

  It follows that: $\Delta \vdash_{\mathcal{P}(Var)} \texttt{c}_1\texttt{;c}_2 : X_1 \cap X_2$.

- **if:** `if b then` $\texttt{c}_1$ `else` $\texttt{c}_2$.
  The hypothesis ensures that there are two sets of variables $X_1, X_2 \subseteq Var$ such that:

  $$\Delta^*(\bigcup_{v \in vars(b)} \Gamma_0(v)) \subseteq X_1 \cap X_2$$
  $$\Gamma_0 \vdash_{\mathcal{L}_{\Delta^*}} \texttt{c}_1 : X_1$$
  $$\Gamma_0 \vdash_{\mathcal{L}_{\Delta^*}} \texttt{c}_2 : X_2$$

  As such, one may conclude that: $\bigcup_{v \in vars(b)} \Delta(v) \subseteq X_1 \cap X_2$. Applying the induction hypothesis:

  $$\Delta \vdash_{\mathcal{P}(Var)} \texttt{c}_1 : X_1$$
  $$\Delta \vdash_{\mathcal{P}(Var)} \texttt{c}_2 : X_2$$

  It follows that: $\Delta \vdash_{\mathcal{P}(Var)} \texttt{if b then c}_1 \texttt{ else c}_2 : X_1 \cap X_2$.

- **while:** `while b do c`.
  The hypothesis ensures that there is a set of variables $X \subseteq Var$ such that:

  $$\Delta^*(\bigcup_{v \in vars(b)} \Gamma_0(v)) \subseteq X$$
  $$\Gamma_0 \vdash_{\mathcal{L}_{\Delta^*}} \texttt{c} : X$$

  Applying the induction hypothesis: $\Delta \vdash_{\mathcal{P}(Var)} \texttt{c} : X$ and noticing that:

  $$\Delta^*(\bigcup_{v \in vars(b)} \Gamma_0(v)) \subseteq X \Leftrightarrow \bigcup_{v \in vars(b)} \Delta(v) \subseteq X$$

  the result follows.

□

6 Connecting the Analysis

Given a security lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ and a security labelling Γ, define the following functions:

$$\begin{array}{ll} \alpha_\Gamma : \mathcal{P}(Var) \to \mathcal{L} & \alpha_\Gamma(X) = \bigsqcup\{\Gamma(x) \mid x \in X\} \\ \gamma_\Gamma : \mathcal{L} \to \mathcal{P}(Var) & \gamma_\Gamma(l) = \{x \mid \Gamma(x) \sqsubseteq l\} \end{array} \tag{16}$$

Adopting a confidentiality point of view, given a set of variables $X \subseteq Var$ and a confidentiality level $l \in L$, $\alpha_\Gamma(X)$ denotes the security level at which all variables in X are visible, whereas $\gamma_\Gamma(l)$ denotes the set that comprises all variables that are visible at level l.

Definition 13 (Galois Connection)**.** *Given two lattices $\mathcal{L} = (L, \sqsubseteq_\mathcal{L})$ and $\mathcal{M} = (M, \sqsubseteq_\mathcal{M})$ and two functions $\alpha : L \to M$ and $\gamma : M \to L$, the tuple (L, α, β, M) is said to be a Galois Connection between $\mathcal{L}$ and $\mathcal{M}$ if the following holds:*

- *α and γ are monotonic.*
- *$\forall_{l \in L} \,.\, l \sqsubseteq_\mathcal{L} \gamma \circ \alpha(l)$*
- *$\forall_{m \in M} \,.\, \alpha \circ \gamma(m) \sqsubseteq_\mathcal{M} m$*

Lemma 23 states that α_Γ and γ_Γ form a Galois Connection [7] and Lemmas 24 and 25 state some useful properties of this particular Galois connection that will be used in later sections.

Lemma 23. *Given an arbitrary security lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ and a security labelling $\Gamma : Var \to L$, $(\mathcal{P}(Var), \alpha_\Gamma, \gamma_\Gamma, \mathcal{L})$ where α_Γ and γ_Γ are defined as above is a Galois connection.*

Proof.

- Monotonocity of α_Γ and γ_Γ. Consider two arbitrary sets $X_1, X_2 \subseteq Var$ and suppose that $X_1 \subseteq X_2$, then it is clear that $\alpha_\Gamma(X_1) \sqsubseteq \alpha_\Gamma(X_2)$. Consider two arbitrary security levels $l_1, l_2 \in L$ and suppose $l_1 \sqsubseteq l_2$, then it follows that $\gamma_\Gamma(l_1) \subseteq \gamma_\Gamma(l_2)$.
- Consider an arbitrary set of variables $X \subseteq Var$. $\gamma_\Gamma \circ \alpha_\Gamma(X)$ can be rewritten in the followin way:

  $$\gamma_\Gamma \circ \alpha_\Gamma(X) = \{x \mid \Gamma(x) \sqsubseteq \bigsqcup\{\Gamma(y) \mid y \in X\}\}$$

  Thus, $X \subseteq \gamma_\Gamma \circ \alpha_\Gamma(X)$.
- Consider an arbitrary security level $l \in L$. $\alpha_\Gamma \circ \gamma_\Gamma(l)$ can be rewritten in the following way:

  $$\alpha_\Gamma \circ \gamma_\Gamma(l) = \bigsqcup\{\Gamma(x) \mid x \in \{y \mid \Gamma(y) \sqsubseteq l\}\}$$

  Thus, $\alpha_\Gamma \circ \gamma_\Gamma(l) \sqsubseteq l$.

□

Lemma 24. *Given a security lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ and a labelling $\Gamma : Var \to L$, $\alpha_\Gamma \circ \gamma_\Gamma : L \to L$ is a reduction operator and $\gamma_\Gamma \circ \alpha_\Gamma : \mathcal{P}(Var) \to \mathcal{P}(Var)$ is a closure operator, where α_Γ and γ_Γ are defined as in equation 16.*

Proof.

- $\gamma_\Gamma \circ \alpha_\Gamma$ is a *closure operator*. Lemma 23 guarantees that $(\mathcal{P}(Var), \alpha_\Gamma, \gamma_\Gamma, \mathcal{L})$ is a Galois connection, so $\gamma_\Gamma \circ \alpha_\Gamma$ is clearly extensive. Moreover it is also monotonic because it is the composition of monotonic functions. With respect to idempotence, it suffices to notice that:

$$\gamma_\Gamma \circ \alpha_\Gamma(\gamma_\Gamma \circ \alpha_\Gamma(l)) = \gamma_\Gamma(\alpha_\Gamma(\gamma_\Gamma(\alpha_\Gamma(l))))$$
$$\gamma_\Gamma \circ \alpha_\Gamma(\gamma_\Gamma \circ \alpha_\Gamma(l)) = \gamma_\Gamma \circ \alpha_\Gamma(l)$$

- $\alpha_\Gamma \circ \gamma_\Gamma$ is a *reduction operator*. A *reduction operator* is a map from a set to itself that is: monotonic, idempotent and reductive. Lemma 23 guarantees that $(\mathcal{P}(Var), \alpha_\Gamma, \gamma_\Gamma, \mathcal{L})$ is a Galois connection, so $\alpha_\Gamma \circ \gamma_\Gamma$ is clearly reductive. Moreover, it also monotonic because it is the composition of monotonic functions. With respect to the third claim it suffices to note that for any Galois connection $\alpha_\Gamma \circ \gamma_\Gamma \circ \alpha_\Gamma = \alpha_\Gamma$, thus:

$$\alpha_\Gamma \circ \gamma_\Gamma(\alpha_\Gamma \circ \gamma_\Gamma(l)) = \alpha_\Gamma(\gamma_\Gamma(\alpha_\Gamma(\gamma_\Gamma(l))))$$
$$\alpha_\Gamma \circ \gamma_\Gamma(\alpha_\Gamma \circ \gamma_\Gamma(l)) = \alpha_\Gamma \circ \gamma_\Gamma(l)$$

□

Lemma 25. *Given a security lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ and a labelling $\Gamma : Var \to L$, α_Γ is an additive functions ang γ_Γ is a multiplicative function, , where α_Γ and γ_Γ are defined as in equation 16.*

Proof.

- α_Γ is additive.

$$\alpha_\Gamma(X_1 \cup X_2) = \bigsqcup\{\Gamma(y) \mid y \in X_1 \cup X_2\}$$
$$\alpha_\Gamma(X_1 \cup X_2) = \bigsqcup\{\Gamma(y) \mid y \in X_1\} \sqcup \bigsqcup\{\Gamma(y) \mid y \in X_2\}$$
$$\alpha_\Gamma(X_1 \cup X_2) = \alpha_\Gamma(X_1) \sqcup \alpha_\Gamma(X_2)$$

- γ_Γ is multiplicative.

$$\gamma_\Gamma(l_1 \sqcap l_2) = \{x \mid \Gamma(x) \sqsubseteq l_1 \sqcap l_2\}$$
$$\gamma_\Gamma(l_1 \sqcap l_2) = \{x \mid \Gamma(x) \sqsubseteq l_1\} \cap \{x \mid \Gamma(x) \sqsubseteq l_2\}$$
$$\gamma_\Gamma(l_1 \sqcap l_2) = \gamma_\Gamma(l_1) \cap \gamma_\Gamma(l_2)$$

□

6.1 Dependency Based Confidentiality Certification

Having explained that establishing a security labelling $\Gamma : Var \to L$ induces a Galois connection between $\mathcal{P}(Var)$ and L, this section investigates how to make use of this connection in order to obtain a necessary and sufficient condition for a program to be deemed secure according to the Volpano *et al* type system.
Given a lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ and a security labelling Γ, define the closure operator $\Pi_\Gamma : \mathcal{P}(Var) \to \mathcal{P}(Var)$, where $\Pi_\Gamma = \gamma_\Gamma \circ \alpha_\Gamma$. Lemma 24 guarantees that Π_Γ is indeed a closure operator. Additionally, define the flow function $\Delta_\Gamma : Var \to \mathcal{P}(Var)$, $\Delta_\Gamma(x) = \gamma_\Gamma \circ \alpha_\Gamma(\{x\})$, for all $x \in Var$. That is, for every variable $x \in Var$:

$$\Delta_\Gamma(x) = \gamma_\Gamma(\alpha_\Gamma(\{x\})) = \{y \in Var \mid \Gamma(y) \sqsubseteq \Gamma(x)\} \tag{17}$$

Naturally, since Π_Γ is not necessarily additive, it follows that $\Delta^*_\Gamma \sqsubseteq \Pi_\Gamma$. Δ_Γ plays a central role in certifying that a given program abides by a given confidentiality policy as demonstrated in lemma 28, which states that a given a security labelling Γ types a program with respect to an arbitrary security lattice $\mathcal{L}$ if and only if the corresponding dependency function Δ_Γ types the program with respect to the lattice $\mathcal{P}(Var)$.

Lemma 26. *Given an arbitrary security lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$, a program c, a security labelling $\Gamma : Var \to L$ and a security level $l \in L$ such that $\Gamma \vdash_{\mathcal{L}} c : l$, then:*

$$\Delta_\Gamma \vdash_{\mathcal{P}(Var)} c : \gamma_\Gamma(l)$$

Proof. The proof proceeds by induction on the derivation of $\Gamma \vdash_{\mathcal{L}} c : l$.

- **skip:** `skip`.
  $\Gamma \vdash_{\mathcal{L}} \texttt{skip} : \top$.
  $\Delta_\Gamma \vdash_{\mathcal{P}(Var)} \texttt{skip} : Var$.
  $\gamma_\Gamma(\top) = \{y \mid \Gamma(y) \sqsubseteq \top\} = Var$.

- **assign:** `x := e`.
  Since $\Gamma \vdash_{\mathcal{L}} \texttt{x:=e} : \Gamma(x)$, it follows that:

  $$\begin{array}{l} \bigsqcup_{v \in vars(e)} \Gamma(v) \sqsubseteq \Gamma(x) \\ \forall_{v \in vars(e)} \cdot \alpha_\Gamma(v) \sqsubseteq \alpha_\Gamma(x) \\ \forall_{v \in vars(e)} \cdot \gamma_\Gamma(\alpha_\Gamma(v)) \subseteq \gamma_\Gamma(\alpha_\Gamma(x)) \\ \forall_{v \in vars(e)} \cdot \Delta_\Gamma(v) \subseteq \Delta_\Gamma(x)) \\ \bigcup_{v \in vars(e)} \Delta_\Gamma(v) \subseteq \Delta_\Gamma(x) \end{array}$$

  As such: $\Delta_\Gamma \vdash_{\mathcal{P}(Var)} \texttt{x := e} : \Delta_\Gamma(x)$.
  Additionally, the part of the claim which concerns the writing effect follows since $l = \Gamma(x), \gamma_\Gamma(l) = \Delta_\Gamma(x)$.

- **seq:** $\texttt{c}_1\texttt{;c}_2$.
  The hypothesis ensures that there must be two security levels $l_1, l_2 \in L$ such that:

  $$\Gamma \vdash_{\mathcal{L}} \texttt{c}_1 : l_1 \quad \Gamma \vdash_{\mathcal{L}} \texttt{c}_2 : l_2$$

Applying the induction hypothesis:

$$\Delta_\Gamma \vdash_{\mathcal{P}(Var)} \mathtt{c}_1 : \gamma_\Gamma(l_1) \quad \Delta_\Gamma \vdash_{\mathcal{P}(Var)} \mathtt{c}_2 : \gamma_\Gamma(l_2)$$

Since γ_Γ is multiplicative (lemma 25), it follows that $\gamma_\Gamma(l_1) \cap \gamma_\Gamma(l_2) = \gamma_\Gamma(l_1 \sqcap l_2)$. Thus:

$$\Delta_\Gamma \vdash_{\mathcal{P}(Var)} \mathtt{c}_1;\mathtt{c}_2 : \gamma_\Gamma(l_1 \sqcap l_2)$$

- **if:** `if b then c1 else c2`.
  The hypothesis ensures that there must be two security levels $l_1, l_2 \in L$ such that:

$$\begin{array}{l} \bigsqcup_{v \in vars(b)} \Gamma(v) \sqsubseteq l_1 \sqcap l_2 \\ \Gamma \vdash_{\mathcal{L}} \mathtt{c}_1 : l_1 \\ \Gamma \vdash_{\mathcal{L}} \mathtt{c}_2 : l_2 \end{array}$$

  Applying the induction hypothesis, it follows that:

$$\Delta_\Gamma \vdash_{\mathcal{P}(Var)} \mathtt{c}_1 : \gamma_\Gamma(l_1) \quad \Delta_\Gamma \vdash_{\mathcal{P}(Var)} \mathtt{c}_2 : \gamma_\Gamma(l_2)$$

  Noting that:

$$\begin{array}{l} \bigsqcup_{v \in vars(b)} \Gamma(v) \sqsubseteq l_1 \sqcap l_2 \\ \forall_{v \in vars(b)} \,.\, \Gamma(v) \sqsubseteq l_1 \sqcap l_2 \\ \forall_{v \in vars(b)} \,.\, \alpha_\Gamma(v) \sqsubseteq l_1 \sqcap l_2 \\ \forall_{v \in vars(b)} \,.\, \gamma_\Gamma(\alpha_\Gamma(v)) \subseteq \gamma_\Gamma(l_1 \sqcap l_2) \\ \forall_{v \in vars(b)} \,.\, \Delta_\Gamma(v) \subseteq \gamma_\Gamma(l_1 \sqcap l_2) = \gamma_\Gamma(l_1) \cap \gamma_\Gamma(l_2) \\ \bigcup_{v \in vars(b)} \Delta_\Gamma(v) \subseteq \gamma_\Gamma(l_1 \sqcap l_2) \end{array}$$

  One may conclude: $\Delta_\Gamma \vdash_{\mathcal{P}(Var)} \mathtt{c}_1;\mathtt{c}_2 : \gamma_\Gamma(l_1 \sqcap l_2)$

- **while:** `while b do c`.
  The hypothesis guarantees that there is a security level $l \in L$ such that:

$$\begin{array}{l} \bigsqcup_{v \in vars(b)} \Gamma(v) \sqsubseteq l \\ \Gamma \vdash_{\mathcal{L}} \mathtt{c} : l \end{array}$$

  Applying the induction hypothesis, it follows that: $\Delta_\Gamma \vdash_{\mathcal{P}(Var)} c : \gamma_\Gamma(l)$. Observing that:

$$\begin{array}{l} \bigsqcup_{v \in vars(b)} \Gamma(v) \sqsubseteq l \\ \forall_{v \in vars(b)} \,.\, \Gamma(v) \sqsubseteq l \\ \forall_{v \in vars(b)} \,.\, \alpha_\Gamma(v) \sqsubseteq l \\ \forall_{v \in vars(b)} \,.\, \Delta_\Gamma(v) = \gamma_\Gamma(\alpha_\Gamma(v)) \subseteq \gamma_\Gamma(l) \\ \bigcup_{v \in vars(b)} \Delta_\Gamma(v) \subseteq \gamma_\Gamma(l) \end{array}$$

  It follows that: $\Delta_\Gamma \vdash_{\mathcal{P}(Var)} \mathtt{while\ b\ do\ c} : \gamma_\Gamma(l)$.

□

Lemma 27. *Given an arbitrary security lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$, a program c, a security labelling $\Gamma : Var \to L$ and a set of variables $X \subseteq Var$ such that: $\Delta_\Gamma \vdash_{\mathcal{P}(Var)} \mathtt{c} : X$. Then, there is a security level $l \in L$ such that: $\Gamma \vdash_{\mathcal{L}} c : l$, where $\alpha_\Gamma(X) \sqsubseteq l$.*

Proof. The proof proceeds by induction on the derivation of $\Delta_\Gamma \vdash_{\mathcal{P}(Var)} c : X$.

- **skip:** `skip`.
  $\Delta_\Gamma \vdash_{\mathcal{P}(Var)} \texttt{skip} : Var$.
  $\Gamma \vdash_{\mathcal{L}} \texttt{skip} : \top$.
  It is always the case that $\alpha_\Gamma(Var) \sqsubseteq \top$.

- **assign:** $\texttt{x} := \texttt{e}$.
  The hypothesis ensures that $\Delta_\Gamma \vdash_{\mathcal{P}(Var)} \texttt{x} := \texttt{e} : \Delta_\Gamma(x)$. Thus, it follows that:

  $$\begin{array}{l} \forall_{v \in vars(e)} \,.\, \Delta_\Gamma(v) \subseteq \Delta_\Gamma(x) \\ \forall_{v \in vars(e)} \,.\, \Gamma(v) \sqsubseteq \Gamma(x) \\ \bigsqcup_{v \in vars(e)} \,.\, \Gamma(v) \sqsubseteq \Gamma(x) \end{array}$$

  With respect to the writing effect it suffices to note that:

  $$\begin{array}{rcl} \alpha_\Gamma(\Delta_\Gamma(x)) & = & \alpha_\Gamma(\gamma_\Gamma(\alpha_\Gamma(x))) \\ & = & \alpha_\Gamma(x) \\ & = & \Gamma(x) \end{array}$$

- **seq:** $\texttt{c}_1;\texttt{c}_2$.
  The hypothesis ensures that there are two sets of variables $X_1, X_2 \subseteq Var$ such that:

  $$\Delta_\Gamma \vdash_{\mathcal{P}(Var)} \texttt{c}_1 : X_1 \quad \Delta_\Gamma \vdash_{\mathcal{P}(Var)} \texttt{c}_2 : X_2$$

  Applying the induction hypothesis, one may conclude that there are two security levels $l_1, l_2 \in L$ such that:

  $$\begin{array}{ll} \alpha_\Gamma(X_1) \sqsubseteq l_1 & \Gamma \vdash_{\mathcal{L}} \texttt{c}_1 : l_1 \\ \alpha_\Gamma(X_2) \sqsubseteq l_2 & \Gamma \vdash_{\mathcal{L}} \texttt{c}_2 : l_2 \end{array}$$

  It follows that: $\Gamma \vdash_{\mathcal{L}} \texttt{c}_1 : l_1 \sqcap l_2$.
  Since α_Γ is monotonic (lemma 23), $\alpha_\Gamma(X_1 \cap X_2) \sqsubseteq \alpha_\Gamma(X_1) \sqcap \alpha_\Gamma(X_2)$ and thus $\alpha_\Gamma(X_1 \cap X_2) \sqsubseteq l_1 \sqcap l_2$.

- **if:** `if b then` $\texttt{c}_1$ `else` $\texttt{c}_2$.
  The hypothesis ensures that there are two sets of variables $X_1, X_2 \subseteq Var$ such that:

  $$\begin{array}{l} \bigcup_{v \in vars(b)} \Delta_\Gamma(v) \subseteq X_1 \cap X_2 \\ \Delta_\Gamma \vdash_{\mathcal{P}(Var)} \texttt{c}_1 : X_1 \\ \Delta_\Gamma \vdash_{\mathcal{P}(Var)} \texttt{c}_2 : X_2 \end{array}$$

  Apllying the induction hypothesis, one may conclude that there are two security levels $l_1, l_2 \in L$ such that:

  $$\begin{array}{ll} \alpha_\Gamma(X_1) \sqsubseteq l_1 & \Gamma \vdash_{\mathcal{L}} \texttt{c}_1 : l_1 \\ \alpha_\Gamma(X_2) \sqsubseteq l_2 & \Gamma \vdash_{\mathcal{L}} \texttt{c}_2 : l_2 \end{array}$$

Thus:

$$\begin{array}{l}\bigcup_{v\in vars(b)} \Delta_\Gamma(v) \subseteq X_1 \sqcap X_2\\ \forall_{v\in vars(b)} \,.\, \Delta_\Gamma(v) = \gamma_\Gamma(\alpha_\Gamma(\{v\})) \subseteq X_1 \sqcap X_2\\ \forall_{v\in vars(b)} \,.\, \Gamma(v) = \alpha_\Gamma(\{v\}) \sqsubseteq \alpha_\Gamma(X_1 \sqcap X_2)\\ \forall_{v\in vars(b)} \,.\, \Gamma(v) \sqsubseteq \alpha_\Gamma(X_1) \sqcap \alpha_\Gamma(X_2)\\ \forall_{v\in vars(b)} \,.\, \Gamma(v) \sqsubseteq l_1 \sqcap l_2\\ \bigsqcup_{v\in vars(b)} \Gamma(v) \sqsubseteq l_1 \sqcap l_2\end{array}$$

One may therefore conclude that: $\Gamma \vdash_{\mathcal{L}} \texttt{if b then c}_1 \texttt{ else c}_2 : l_1 \sqcap l_2$. With respect to the claim regrading the writing effect, it suffices to apply the same reasoning as in [seq].

- **while:** `while b do c`.
  The hypothesis ensures that there are a set of variables $X \subseteq Var$ such that:

$$\begin{array}{l}\bigcup_{v\in vars(b)} \Delta_\Gamma(v) \subseteq X\\ \Delta_\Gamma \vdash_{\mathcal{P}(Var)} \texttt{c} : X\end{array}$$

  The induction hypothesis guarantees that there is a security level $l \in L$ such that:

$$\alpha_\Gamma(X) \sqsubseteq l \quad \Gamma \vdash_{\mathcal{L}} c : l$$

  Thus:

$$\begin{array}{l}\bigcup_{v\in vars(b)} \Delta_\Gamma(v) \subseteq X\\ \forall_{v\in vars(b)} \,.\, \Delta_\Gamma(v) \subseteq X\\ \forall_{v\in vars(b)} \,.\, \Gamma(v) \sqsubseteq \alpha_\Gamma(X)\\ \sqcup_{v\in vars(b)} \Gamma(v) \sqsubseteq l\end{array}$$

  As such it follows that: $\Gamma \vdash_{\mathcal{L}} \texttt{while b do c} : l$.

□

Lemma 28. *Given an arbitrary security lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$, a program c, a security labelling $\Gamma : Var \to L$, then the following equivalence holds:*

$$\exists_{l\in L} \,.\, \Gamma \vdash_{\mathcal{L}} c : l \Leftrightarrow \exists_{X\subseteq Var} \,.\, \Delta_\Gamma \vdash_{\mathcal{P}(Var)} c : X$$

Proof. This lemma is an immediate consequence of lemmas 26 and 27. □

Theorem 2 establishes a sufficient and necessary condition for a program to be deemed secure. Informally, this condition states that a program is typable if and only high variables do not depend on low variables.

Theorem 2. *Given an arbitrary security lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$, a program c, a security labelling $\Gamma : Var \to L$ then the following equivalence holds:*

$$\exists_{l\in L} \,.\, \Gamma \vdash_{\mathcal{L}} c : l \Leftrightarrow \forall_{x\in Var} \,.\, \Gamma(x) = \bigsqcup \{\Gamma(y) \mid y \in \Delta^{opt}(x)\}$$

Where $\Delta^{opt} = \mathcal{D}[\![c]\!](\varnothing, \Gamma_0)$.

Proof. First, note that for any given flow function Δ the following equivalence holds:

$$\exists_{X \subseteq Var} \,.\, \Delta \vdash_{\mathcal{P}(Var)} c : X \Leftrightarrow \Delta^{opt} = \mathcal{D}[[c]](\varnothing, \Gamma_0) \sqsubseteq \Delta$$

For the direct side of this implication, suppose that there is a set of variables $X \subseteq Var$ such that $\Delta \vdash_{\mathcal{P}(Var)} c : X$, then lemma 22 guarantees that $\Gamma_0 \vdash_{\mathcal{L}_{\Delta^*}} c : X$. Thus, applying theorem 1, it follows that $\Delta^{opt} \sqsubseteq \Delta$. Conversely, lemma 15 guarantees that there is a set of variables $X' \subseteq Var$ such that $\Gamma_0 \vdash_{\mathcal{L}_{\Delta^{opt}}} c : X'$. Applying lemma 10, it follows that there is a set of principals $X'' \subseteq P$ such that $X' \subseteq X''$ and $\Gamma_0 \vdash_{\mathcal{L}_\Delta} c : X''$. Thus, applying lemma 22 it follows that $\Delta \vdash_{\mathcal{P}(P)} c : X''$.
Lemma 28 states that:

$$\exists_{l \in L} \,.\, \Gamma \vdash_{\mathcal{L}} c : l \Leftrightarrow \exists_{X \subseteq Var} \,.\, \Delta_\Gamma \vdash_{\mathcal{P}(Var)} c : X$$

which, considering the equivalence established above entails the following:

$$\exists_{l \in L} \,.\, \Gamma \vdash_{\mathcal{L}} c : l \Leftrightarrow \Delta^{opt} = \mathcal{D}[[c]](\varnothing, \Gamma_0) \sqsubseteq \Delta_\Gamma$$

Clearly:

$$\begin{array}{l} \Delta^{opt} \sqsubseteq \Delta_\Gamma \Leftrightarrow \forall_{x \in X} \,.\, \Delta^{opt}(x) \subseteq \Delta_\Gamma(x) \\ \Delta^{opt} \sqsubseteq \Delta_\Gamma \Leftrightarrow \forall_{x \in X} \,.\, \Delta^{opt}(x) \subseteq \{y \mid \Gamma(y) \sqsubseteq \Gamma(x)\} \\ \Delta^{opt} \sqsubseteq \Delta_\Gamma \Leftrightarrow \forall_{x \in X} \,.\, \Gamma(x) = \bigsqcup\{\Gamma(y) \mid y \in \Delta^{opt}(x)\} \end{array}$$

□

7 Inferring security labellings

This section considers the problem of, given a security lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$, a security labelling $\Gamma_i : Var \to L$ and a program c, computing the lowest security labelling Γ such that:

$$\begin{array}{c} \Gamma_i \sqsubseteq \Gamma \Leftrightarrow \forall_{v \in Var} \,.\, \Gamma_i(v) \sqsubseteq \Gamma(v) \\ \exists l \in L.\ \Gamma \vdash_L c : l \end{array}$$

Note that for any Γ_i and program c, the security labelling which assigns every variable to the highest security level verifies the two conditions stated above. Naturally, the goal is to find the smallest one. Observe that when equipped with pointwise subset inclusion as its partial order, the set of security labellings that map a given finite set of variables to a given security lattice is a complete lattice.

This problem was already approached by Hunt *et al* in [13] using flow sensitive type systems, where a *fixed labelling* is computed, but the program is modified in such a way that each variable is assigned only once.

7.1 A calculus of fixed types

Given any two security labellings $\Gamma_1, \Gamma_2 : Var \to L$, definition 14 establishes a condition that, when verified, guarantees that all programs typed under Γ_1 are also typed under Γ_2.

Definition 14 (Type Subsumption)**.** *Given two security labellings* $\Gamma_1, \Gamma_2 : Var \to L$*,* Γ_1 *is said to subsume* Γ_2 *if the following holds:*

$$\forall_{u,v \in Var} \,.\, \Gamma_1(u) \sqsubseteq \Gamma_1(v) \Rightarrow \Gamma_2(u) \sqsubseteq \Gamma_2(v)$$

Lemma 29. *Given a security lattice* $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$*, a program* c*, two security labellings* $\Gamma_1, \Gamma_2 : Var \to L$ *and a security level* $l_1 \in L$ *such that* $\Gamma_1 \vdash_{\mathcal{L}} c : l_1$ *and* Γ_1 *subsumes* Γ_2*, then there is a security level* $l_2 \in L$ *such that* $l_1 \sqsubseteq l_2$*,* $\Gamma_2 \vdash_{\mathcal{L}} c : l_2$ *and* $\forall_{v \in Var} \,.\, \Gamma_1(v) \sqsubseteq l_1 \Rightarrow \Gamma_2(v) \sqsubseteq l_2$*.*

Proof. The proof proceeds by induction on the structure of c.

- **skip:** `skip`.
  $\Gamma_2 \vdash \texttt{skip} : \top$.
  $\top \sqsubseteq \top$.
  $\forall_{v \in Var} \,.\, \Gamma(v) \sqsubseteq \top \Rightarrow \forall_{v \in Var} \,.\, \Gamma(v) \sqsubseteq \top$.

- **assign:** `x := e`.
  The hypothesis guarantees that $\Gamma_1 \vdash \texttt{x} := \texttt{e} : l_1$, so:

  $$\begin{array}{l} \bigsqcup_{v \in vars(e)} \Gamma_1(v) \sqsubseteq \Gamma_1(x) \\ \forall_{v \in vars(e)} \Gamma_1(v) \sqsubseteq \Gamma_1(x) \\ \forall_{v \in vars(e)} \Gamma_2(v) \sqsubseteq \Gamma_2(x) \\ \bigsqcup_{v \in vars(e)} \Gamma_2(v) \sqsubseteq \Gamma_2(x) \end{array}$$

  Thus: $\Gamma_2 \vdash \texttt{x} := \texttt{e} : \Gamma_2(x)$. Clearly $l_1 \sqsubseteq l_2$, since $\Gamma_1(x) \sqsubseteq \Gamma_2(x)$. Additionally, suppose that for a variable $v \in Var$, $\Gamma(v) \sqsubseteq l_1$, the hypothesis ensures that $\Gamma(v) \sqsubseteq l_2$ (since Γ_1 subsumes Γ_2).

- **seq:** `c1;c2`.
  The hypothesis ensures that there are two security levels $l_1, l_2 \in L$ such that $\Gamma_1 \vdash \texttt{x} := \texttt{e} : l_1$ and $\Gamma_2 \vdash \texttt{x} := \texttt{e} : l_2$. Thus applying the induction hypothesis, we conclude that there must be two security levels $l_1', l_2' \in L$ such that:

  $$\begin{array}{llll} l_1 \sqsubseteq l_1' & \Gamma_2 \vdash c_1 : l_1' & \forall_{v \in Var} \,.\, \Gamma_1(v) \sqsubseteq l_1 \Rightarrow \Gamma_2(v) \sqsubseteq l_1' \\ l_2 \sqsubseteq l_2' & \Gamma_2 \vdash c_2 : l_2' & \forall_{v \in Var} \,.\, \Gamma_1(v) \sqsubseteq l_2 \Rightarrow \Gamma_2(v) \sqsubseteq l_2' \end{array}$$

  It thus follows that:

  $$l_1 \sqcap l_2 \sqsubseteq l_1' \sqcap l_2' \quad \Gamma_2 \vdash c_1; c_2 : l_1' \sqcap l_2' \quad \forall_{v \in Var} \,.\, \Gamma_1(v) \sqsubseteq l_1 \sqcap l_2 \Rightarrow \Gamma_2(v) \sqsubseteq l_1' \sqcap l_2'$$

- **if:** `if b then c1 else c2`.
  The hypothesis ensures that there are two security levels $l_1, l_2 \in L$ such that $\Gamma_1 \vdash c_1 : l_1$, $\Gamma_1 \vdash c_2 : l_2$ and $\bigsqcup_{v \in vars(b)} \Gamma_1(v) \sqsubseteq l_1 \sqcap l_2$. Applying the induction hypothesis:

  $$\begin{array}{llll} l_1 \sqsubseteq l_1' & \Gamma_2 \vdash c_1 : l_1' & \forall_{v \in Var} \,.\, \Gamma_1(v) \sqsubseteq l_1 \Rightarrow \Gamma_2(v) \sqsubseteq l_1' \\ l_2 \sqsubseteq l_2' & \Gamma_2 \vdash c_2 : l_2' & \forall_{v \in Var} \,.\, \Gamma_1(v) \sqsubseteq l_2 \Rightarrow \Gamma_2(v) \sqsubseteq l_2' \end{array}$$

Observing that:

$$\begin{array}{l}\bigsqcup_{v\in vars(b)}\Gamma_1(v)\sqsubseteq l_1\sqcap l_2\\ \forall_{v\in vars(b)}\ .\ \Gamma_1(v)\sqsubseteq l_1\wedge\Gamma_1(v)\sqsubseteq l_2\\ \forall_{v\in vars(b)}\ .\ \Gamma_2(v)\sqsubseteq l_1'\wedge\Gamma_2(v)\sqsubseteq l_2'\bigsqcup_{v\in vars(b)}\Gamma_2(v)\sqsubseteq l_1'\sqcap l_2'\end{array}$$

As such: $\Gamma_2\vdash$ `if b then c1 else c2` $: l_1'\sqcap l_2'$. The remaining two claims follow in the same way as in [seq].

- **while:** `while b do c`.
  The hypothesis guarantees that there is a security level $l_1\in L$ such that: $\Gamma_1\vdash c: l_1$ and $\bigsqcup_{v\in vars(b)}\Gamma(v)\sqsubseteq l_1$. Applying the induction hypothesis, one may conclude that there is a security level $l_2\in L$ such that:

  $$l_1\sqsubseteq l_2\quad \Gamma_2\vdash c: l_2\quad \forall_{v\in Var}\ .\ \Gamma_1(v)\sqsubseteq l_1\Rightarrow\Gamma_2(v)\sqsubseteq l_2$$

  Noting that:

  $$\begin{array}{l}\bigsqcup_{v\in vars(b)}\Gamma_1(v)\sqsubseteq l_1\\ \forall_{v\in vars(b)}\ .\ \Gamma_1(v)\sqsubseteq l_1\\ \forall_{v\in vars(b)}\ .\ \Gamma_2(v)\sqsubseteq l_2\\ \bigsqcup_{v\in vars(b)}\Gamma_2(v)\sqsubseteq l_2\end{array}$$

  It follows that $\Gamma_2\vdash$ `while b do c` $: l_2$. The other two claims are immediately implied by the induction hypothesis.

□

When presented the problem specified in the begining of this section, lemma 29 can be easily used to conceive an analysis in order to find a security labelling which is both greater than the one given as input and also types the targeted program. This analysis is presented in definition 15.

Definition 15 (A calculus of fixed types)**.**

`SKIP` $\quad \mathcal{T}[[\texttt{skip}]](l,\Gamma) = \Gamma$

`ASSIGN`
$$\mathcal{T}[[\texttt{x := e}]](l,\Gamma) = \Gamma'$$
$$\Gamma'(y) = \begin{cases} \Gamma(y) & \text{if } \Gamma(x) \not\sqsubseteq \Gamma(y) \\ \Gamma(y) \sqcup l \sqcup \left(\bigsqcup_{v \in vars(e)} \Gamma(v)\right) & \text{if } \Gamma(x) \sqsubseteq \Gamma(y) \end{cases}$$

`SEQ` $\quad \mathcal{T}[[\texttt{c}_1\texttt{;c}_2]](l,\Gamma) = \mathcal{D}[[c_2]](l, \mathcal{T}[[c_1]](l,\Gamma))$

`IF`
$$\mathcal{T}[[\texttt{if b then c}_1\texttt{ else c}_2]](l,\Gamma) = \mathcal{T}[[c_2]](l_b, \mathcal{T}[[c_1]](l_b,\Gamma))$$
$$l_b = \bigsqcup_{v \in vars(b)} \Gamma(v) \sqcup l$$

`WHILE`
$$\mathcal{T}[[\texttt{while b do c}]](l,\Gamma) = \bigsqcup\{H^k(\bot) \mid 0 \leq k\}$$
Where:
$$H(\Gamma'') = \mathcal{D}(\bigsqcup_{v \in vars(b)} \Gamma(''v) \sqcup l, \Gamma'') \sqcup \Gamma$$

It is important to note that the calculus of fixed types presented in Definition 15 does not use the usual least fixed point operator for the [WHILE] rule. Instead, it declares explicitly how to construct the intended fixed point. The reason for doing this is that this calculus of fixed types is not monotone (which is illustrated in example 4). As such, constructing a fixed point *à Kleene* may not yield the least fixed point. Nevertheless, it surely yields a fixed point (since the analysis is extensive Lemma 30).

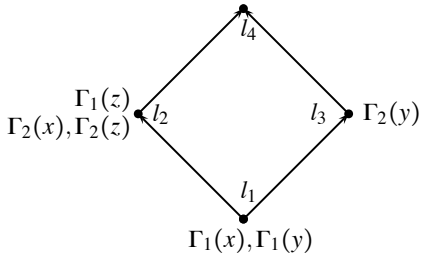
Example 4 (The analysis is not monotone)**.** *This example considers a program $x := z$, a security lattice over the set $L = \{l_1, l_2, l_3, l_4\}$ (which is depicted below) and two security labellings:*

$$\Gamma_1 = [x \mapsto l_1, y \mapsto l_1, z \mapsto l_2] \quad \Gamma_2 = [x \mapsto l_2, y \mapsto l_3, z \mapsto l_2]$$

Clearly $\Gamma_1 \sqsubseteq \Gamma_2$. Applying the proposed calculus of fixed types to these security labellings yields the following results:

$$\Gamma_1' = [x \mapsto l_2, y \mapsto l_2, z \mapsto l_2] \quad \Gamma_2' = [x \mapsto l_2, y \mapsto l_3, z \mapsto l_2]$$

However, $\Gamma_1' \not\sqsubseteq \Gamma_2'$.



Despite not being monotone this analysis is well-defined because it is extensive and the lattice of security labellings considered is finite. The existence of the analysis proved in Lemma 30.

Lemma 30 (Existence and Extensiveness)**.** *Given a program c and an arbitrary security lattice* $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$*, the corresponding calculus of fixed typed* $\mathcal{T}[[c]]$ *then:*

- $\mathcal{T}[[c]]$ *is extensive in its second argument. That is, for any security level* $l \in L$ *and security labelling* $\Gamma : Var \to L$*,* $\Gamma \sqsubseteq \mathcal{T}[[c]](l, \Gamma)$*.*
- *For any security labelling* $\Gamma : Var \to L$ *and security level* $l \in L$*,* $\mathcal{T}[[c]]$ *is well-defined.*

Proof. The proof proceeds by induction on the structure of c.

- **skip:** `skip`.
  $\mathcal{T}[[\texttt{skip}]](l, \Gamma) = \Gamma$. The three claims hold.
- **assign:** `x := e`.
  $\mathcal{T}[[\texttt{x := e}]](l, \Gamma) = \Gamma'$. It is clear that Γ' is uniquely defined and that for every variable $x \in Var$, $\Gamma(x) \sqsubseteq \Gamma'(x)$.
- **seq:** $\texttt{c}_1\texttt{;c}_2$.
  $\mathcal{T}[[\texttt{c}_1\texttt{;c}_2]](l, \Gamma) = \mathcal{T}[[c_2]](l, \mathcal{T}[[c_1]](l, \Gamma))$. Applying the induction hypothesis, it follows that $\mathcal{T}[[c_1]](l, \Gamma)$ is uniquely defined and that $\Gamma \sqsubseteq \mathcal{T}[[c_1]](l, \Gamma)$. Applying the induction hypothesis again yiels:

  $$\mathcal{T}[[c_1]](l, \Gamma) \sqsubseteq \mathcal{T}[[c_2]](l, \mathcal{T}[[c_1]](l, \Gamma))$$

  Moreover, the induction hypothesis guarantees the uniqueness of Γ'. Thus, by transitivity it follows that $\Gamma \sqsubseteq \Gamma'$.
- **if:** $\texttt{if b then c}_1\texttt{ else c}_2$.
  Consider a given security level l and a given security labelling Γ, applying the induction hypothesis yields:

  $$\Gamma \sqsubseteq \Gamma' = \mathcal{T}[[c_1]](l \sqcup (\bigsqcup_{v \in vars(b)} \Gamma(v)), \Gamma)$$

Addtionally it guarantees that Γ' is well defined. Applying the induction hypothesis yet again:

$$\Gamma' \sqsubseteq \Gamma'' = \mathcal{T}[[c_2]](l \sqcup (\bigsqcup_{v \in vars(b)} \Gamma(v)), \Gamma')$$

And Γ'' is well defined.

- **while:** `while b do c`.
  Consider an arbitrary security labelling Γ and a security level $l \in L$, one must prove that for all $k \geq 1$, $\Gamma \sqsubseteq H^k(\bot)$, where $H(\Gamma) = \mathcal{F}[[c]](l \sqcup \bigsqcup_{v \in vars(b)} \Gamma(b), \Gamma)$. The proof proceeds by induction on k:
  - $k = 1$. Applying the outer induction hypothesis the result follows immediately.
  - $k + 1$. $H^{k+1}(\bot) = H(H^k(\bot))$. Applying the inner induction hypothesis it follows that $H^k(\bot)$ is extensive. Thus, applying the outer induction hypothesis, the result follows.

  As such, $\{H^k(\bot) \mid k \geq 0\}$ is an ascending chain, since the lattice of security labellings is finite, one may also conclude that this chain is finite.

□

Lemma 31. *For every security labelling Γ, every security level l and program c if $\mathcal{T}[[c]](l, \Gamma) = \Gamma'$, then Γ subsumes Γ'.*

Proof. The proof proceeds by induction on the struture of c.

- **skip:** `skip`.
  $\mathcal{T}[[\texttt{skip}]](l, \Gamma) = \Gamma$. Naturally, Γ subsumes itself.
- **assign:** `x := e`.
  $\mathcal{T}[[\texttt{x} := \texttt{e}]](l, \Gamma) = \Gamma'$. Consider two variables $u, v \in Var$ such that $\Gamma(u) \sqsubseteq \Gamma(v)$. There are two cases two consider:
  - $\Gamma(x) \not\sqsubseteq \Gamma(u)$. It follows that $\Gamma'(u) = \Gamma(u)$. Lemma 30 ensures that $\Gamma(v) \sqsubseteq \Gamma'(v)$, so: $\Gamma'(u) \sqsubseteq \Gamma'(v)$.
  - $\Gamma(x) \sqsubseteq \Gamma(u)$. The hypothesis ensures that $\Gamma(x) \sqsubseteq \Gamma(v)$. Thus:

$$\Gamma'(u) = \Gamma(u) \sqcup l \sqcup \left(\bigsqcup_{v \in vars(e)} \Gamma(v) \right) \sqsubseteq \Gamma(v) \sqcup l \sqcup \left(\bigsqcup_{v \in vars(e)} \Gamma(v) \right) = \Gamma'(v)$$

- **seq:** `c1;c2`.
  $\mathcal{T}[[c_1]](l, \Gamma) = \Gamma'$ and $\mathcal{T}[[c_2]](l, \Gamma') = \Gamma''$. Applying the induction hypothesis, it follows that Γ subsumes Γ' and Γ' subsumes Γ'', which implies that Γ subsumes Γ'':

$$\begin{aligned}
&\Gamma(u) \sqsubseteq \Gamma(v) \Rightarrow \Gamma'(u) \sqsubseteq \Gamma'(v) \quad \text{(Since } \Gamma \text{ subsumes } \Gamma') \\
&\Gamma'(u) \sqsubseteq \Gamma'(v) \Rightarrow \Gamma''(u) \sqsubseteq \Gamma''(v) \quad \text{(Since } \Gamma' \text{ subsumes } \Gamma'') \\
&\Gamma(u) \sqsubseteq \Gamma(v) \Rightarrow \Gamma''(u) \sqsubseteq \Gamma''(v) \quad \text{(By transitivity)}
\end{aligned}$$

- **if:** `if b then c1 else c2`.
  $\Gamma' = \mathcal{T}[[c_1]](l \sqcup (\bigsqcup_{v \in vars(b)} \Gamma(v)), \Gamma)$ and $\Gamma'' = \mathcal{T}[[c_2]](l \sqcup (\bigsqcup_{v \in vars(b)} \Gamma(v)), \Gamma')$. Applying the induction hypothesis, it follows that Γ subsumes Γ' and Γ' subsumes Γ'', which implies that Γ subsumes Γ''.

- **while:**`while b do c`.
  One has to prove that for all $k \geq 0$, Γ subsumes $H^k(\bot)$, where: $H(\Gamma) = \mathcal{F}[[c]](l \sqcup \bigsqcup_{v \in vars(b)} \Gamma(b), \Gamma)$. The proof proceeds by induction on k:

  – k=0. Clearly, Γ subsumes itself.

  – k+1. The interior induction hypothesis guarantees that Γ subsumes $H^k(\bot)$. The outer induction hypothesis guarantess that $H^k(\bot)$ subsumes $H^{k+1}(\bot)$, which implies that Γ subsumes $H^{k+1}(\bot)$.

  Since $\{H^k(\bot) \mid k \geq 0\}$ is an ascending chain, it follows that Γ subsumes its upper bound.

□

Lemma 32 (Soundness)**.** *Given a program c, a security lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$, a security labelling $\Gamma : Var \to L$, and a security level $l \in L$, if $\mathcal{T}[[c]](l, \Gamma) = \Gamma'$, then there is a security level $l' \in L$ such that: $l \sqsubseteq l'$, $\Gamma' \vdash c : l'$ and $\forall_{v \in Var}\ .\ \Gamma(v) \sqsubseteq l \Rightarrow \Gamma'(v) \sqsubseteq l'$*

Proof.

skip: `skip`.
$\mathcal{T}[[\texttt{skip}]](l, \Gamma) = \Gamma$. It is always the case that $\Gamma \vdash \texttt{skip} : \top$ and $l \sqsubseteq \top$. With respect to the third claim it is easy to see that it holds, since $\forall_{v \in Var}\ .\ \Gamma(v) \sqsubseteq \top$.

assign: `x := e`.
$\mathcal{T}[[\texttt{x := e}]](l, \Gamma) = \Gamma'$. Since $\Gamma(x) \sqsubseteq \Gamma(x)$ it follows that: $\Gamma'(x) = \Gamma(x) \sqcup l \sqcup \Gamma(e)$. Additionally, for every variable $v \in vars(e)$, $\Gamma'(v) \sqsubseteq \Gamma(e) \sqcup l$. Thus:

$$\bigsqcup_{v \in vars(e)} \Gamma'(v) \sqsubseteq \Gamma'(x)$$

So, $\Gamma' \vdash c : \Gamma(x)$. Moreover, $l \sqsubseteq \Gamma'(x)$. Suppose that for an arbitrary variable $v \in Var$ $\Gamma(v) \sqsubseteq l$. There are two cases two consider:

- $\Gamma(x) \not\sqsubseteq \Gamma(v)$. In this case, $\Gamma'(v) = \Gamma(v)$. Additionally, since $l \sqsubseteq \Gamma'(x)$, it follows that:
  $$\Gamma'(v) \sqsubseteq l \sqsubseteq \Gamma'(x)$$

- $\Gamma(x) \sqsubseteq \Gamma(v)$. It follows that: $\Gamma(x) \sqcup l = \Gamma(v) \sqcup l = l$. As such:
  $$\Gamma'(x) = \Gamma'(v) = l \sqcup \bigsqcup_{y \in vars(e)} \Gamma(y)$$

seq: $\texttt{c}_1;\texttt{c}_2$.
$\mathcal{T}[[c_1]](l,\Gamma) = \Gamma'$ and $\mathcal{T}[[c_2]](l,\Gamma') = \Gamma''$. Applying the induction hypothesis, there must exist two security levels $l_1, l_2 \in L$ such that:

$$\begin{array}{llll} l \sqsubseteq l_1 & \Gamma' \vdash \texttt{c}_1 : l_1 & \forall_{v \in vars(b)} \,.\, \Gamma(v) \sqsubseteq l \Rightarrow \Gamma'(v) \sqsubseteq l_1 \\ l \sqsubseteq l_2 & \Gamma'' \vdash \texttt{c}_2 : l_2 & \forall_{v \in vars(b)} \,.\, \Gamma'(v) \sqsubseteq l \Rightarrow \Gamma''(v) \sqsubseteq l_2 \end{array}$$

Applying lemma 31, one may conclude that Γ'' subsumes Γ'. So, lemma 29 guarantees that there is a security level $l_1' \in L$ such that:

$$l_1 \sqsubseteq l_1' \quad \Gamma'' \vdash c_1 : l_1' \quad \forall_{v \in Var} \,.\, \Gamma'(v) \sqsubseteq l_1 \Rightarrow \Gamma''(v) \sqsubseteq l_1'$$

which means that:

$$l \sqsubseteq l_1' \sqcap l_2 \quad \Gamma'' \vdash \texttt{c}_1;\texttt{c}_2 : l_1' \sqcap l_2 \quad \forall_{v \in Var} \,.\, \Gamma(v) \sqsubseteq l \Rightarrow \Gamma''(v) \sqsubseteq l_1' \sqcap l_2$$

if: $\texttt{if b then c}_1 \texttt{ else c}_2$.
$\Gamma' = \mathcal{T}[[c_1]](l \sqcup (\bigsqcup_{v \in vars(b)} \Gamma(v)), \Gamma)$ and $\Gamma'' = \mathcal{T}[[c_2]](l \sqcup (\bigsqcup_{v \in vars(b)} \Gamma(v)), \Gamma')$. Applying the induction hypothesis it follows that there are two security levels $l_1, l_2 \in L$ such that:

$$\begin{array}{llll} l \sqcup (\bigsqcup_{v \in vars(b)} \Gamma(v)) \sqsubseteq l_1 & \Gamma' \vdash \texttt{c}_1 : l_1 & \forall_{v \in vars(b)} \,.\, \Gamma(v) \sqsubseteq l \sqcup (\bigsqcup_{v \in vars(b)} \Gamma(v)) \Rightarrow \Gamma'(v) \sqsubseteq l_1 \\ l \sqcup (\bigsqcup_{v \in vars(b)} \Gamma(v)) \sqsubseteq l_2 & \Gamma'' \vdash \texttt{c}_2 : l_2 & \forall_{v \in vars(b)} \,.\, \Gamma(v) \sqsubseteq l \sqcup (\bigsqcup_{v \in vars(b)} \Gamma(v)) \Rightarrow \Gamma''(v) \sqsubseteq l_2 \end{array}$$

Applying lemma 31, one may conclude that Γ'' subsumes Γ'. So, lemma 29 guarantees that there is a security level $l_1' \in L$ such that:

$$l_1 \sqsubseteq l_1' \quad \Gamma'' \vdash c_1 : l_1' \quad \forall_{v \in Var} \,.\, \Gamma'(v) \sqsubseteq l_1 \Rightarrow \Gamma''(v) \sqsubseteq l_1'$$

As such:

$$\begin{array}{ll} l \sqcup (\bigsqcup_{v \in vars(b)} \Gamma(v)) \sqsubseteq l_1' \sqcap l_2 & \Gamma'' \vdash c_1 : l_1' \\ \forall_{v \in vars(b)} \,.\, \Gamma(v) \sqsubseteq l \sqcup (\bigsqcup_{v \in vars(b)} \Gamma(v)) \Rightarrow \Gamma''(v) \sqsubseteq l_1' \sqcup l_2 & \Gamma'' \vdash c_1 : l_2' \end{array}$$

Observing that $\forall_{v \in vars(b)} \,.\, \Gamma(v) \sqsubseteq l \sqcup (\bigsqcup_{v \in vars(b)} \Gamma(v))$, it follows that:

$$\bigsqcup_{v \in vars(b)} \Gamma'(v) \sqsubseteq l_1' \sqcap l_2$$

which entails the result.

while: $\texttt{while b do c}$.
$\Gamma = \mathcal{T}[[c]](l \sqcup \bigsqcup_{v \in vars(b)} \Gamma(v), \Gamma)$. So, applying the induction hypothesis yields:

$$l \sqcup \bigsqcup_{v \in vars(b)} \Gamma(v) \sqsubseteq l' \quad \Gamma' \vdash \texttt{c}_1 : l' \quad \forall_{v \in vars(b)} \,.\, \Gamma(v) \sqsubseteq l \sqcup (\bigsqcup_{v \in vars(b)} \Gamma(v)) \Rightarrow \Gamma'(v) \sqsubseteq l'$$

Observing that $\forall_{v \in vars(b)} \,.\, \Gamma(v) \sqsubseteq l \sqcup (\bigsqcup_{v \in vars(b)} \Gamma(v))$, it follows that: $\Gamma'(b) \sqsubseteq l'$, which implies the result. □

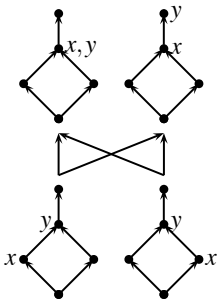
The preceding lemmas (31 and 32) precisely illustrate the way this analysis work: it systematically raises the security labelling Γ so that the new security labelling, Γ', is consistent with the current command being analysed and it ensures that Γ' is consistent with all the commands previously analysed by choosing Γ' in such a way that Γ subsumes Γ'. This fact suggests a new ordering on the security labellings and a possible candidate for an optimality result:

- $\Gamma_1 \leq \Gamma_2$ iff $\Gamma_1 \sqsubseteq \Gamma_2$ and $\forall x, y \in Var.\Gamma_1(x) \sqsubseteq \Gamma_1(y) \Rightarrow \Gamma_2(x) \sqsubseteq \Gamma_2(y)$
- If $\mathcal{T}[[c]](l,\Gamma) = \Gamma'$ and there is a security labelling, Γ'', such that:
  - $\exists l'.\Gamma'' \vdash c : l'$
  - $\Gamma \leq \Gamma''$

  Then: $\Gamma' \leq \Gamma''$

However, this optimality result does not generally hold, since the set of security labellings equipped with the ordering relation defined above is not generally a lattice, which is shown in example 5.

Example 5. *This example illustrates four security labellings over the set of variables $\{x,y\}$. The corresponding Hasse diagram (which orders the security labellings according to the partial order introduced above) is depicted below.*



In the picture each variable is depicted next to the lattice element to which it is mapped. In this example there is no least upper bound, as such it clearly illustrates that the set of security labellings ordered in this way is not generally a lattice.

7.2 A dependency based approach to fixed types computation

Section 7.1 presents a calculus of fixed types and argues that such calculus does not generally finds an optimum solution (with respect to the order of security labellings established). Therefore, before trying to compute the optimal fixed labelling (that is, the lowest one that types the program), one has to discuss its existence, since it is not clear that among all labellings that type a program there is one that is lower than all the

others. This subsection focuses on this problem and establishes a constructive result that proves its existence.

Given an arbitrary security lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$, a labelling $\Gamma : Var \to L$ and a program c, theorem 2 states that c is typable with respect to $\mathcal{L}$ and Γ if and only if:

$$\forall_{x \in Var} \,.\, \Gamma(x) = \bigsqcup \{\Gamma(y) \mid y \in \Delta(x)\}$$

where $\Delta = \mathcal{D}[[c]](\varnothing, \Delta_0)$ and $\Delta_0 : Var \to \mathcal{P}(Var)$ is the map which maps each variable to the singular set that contains it. Therefore, given an arbitrary security labelling Γ_i, the goal is to find the smallest security labelling greater than Γ_i that satisfies the condition expressed above. Consider the operator on security labellings $H : (Var \to L) \to (Var \to L)$ defined in the following way:

$$H(\Gamma) = \bigsqcup \{\Gamma(y) \mid y \in \Delta(x)\} \sqcup \Gamma_i$$

Clearly, the goal is to find the least fixed point of this operator. Since the set of security labellings equipped with pointwise subset inclusion as its partial order is a complete lattice, in order to prove that the least fixed point exists, it is only necessary to prove that H is monotonic. As such, consider two arbitrary security labellings $\Gamma_1, \Gamma_2 : Var \to L$ such that $\Gamma_1 \sqsubseteq \Gamma_2$. For every variable $x \in Var$, $H(\Gamma_1)x \sqsubseteq H(\Gamma_2)x$. Therefore, given an arbitrary security lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$, a security labelling $\Gamma_0 : Var \to L$ and a program c, the least security labelling greater than Γ_i which types the program is:

$$\Gamma^{opt} = \bigsqcup \{H^k(\bot) \mid k \geq 0\}$$

Theorem 3. *Given an arbitrary security lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$, a security labelling $\Gamma_i : Var \to L$ and a program c, the smallest security labelling greater than Γ_i for which there is a security level $l \in L$ such that $\Gamma \vdash_{\mathcal{L}} c : l$ is given by:*

$$\Gamma = \bigsqcup \{H^k(\bot) \mid k \geq 0\}$$

Where $H : (Var \to L) \to (Var \to L)$ is an operator defined as follows:

$$H(\Gamma) = \bigsqcup \{\Gamma(y) \mid y \in \Delta(x)\} \sqcup \Gamma_i$$

Where $\Delta(x) = \mathcal{D}[[c]](\varnothing, \Delta_0)$ and $\Delta_0 : Var \to \mathcal{P}(Var)$ is the map which maps each variable to the singular set that contains it.

8 Computing the strictest confidentiality policy

Given a program c and a security labelling $\Gamma : Var \to L$ (where L is an unstructured set of security levels), the goal of this section is to find the strictest security policy that renders c secure with respect to the Volpano *et al* type system (and hence to trace non-interference). In order to do this, we shall consider that there is always an initial lattice which is known *a priori* and which relates all security levels. This initial lattice corresponds to the strictest security policy which can possibly be enforced and shall be referred to as the *original security lattice*. For instance, when considering security

policies generated by closure operators over a given set of principals, the original lattice corresponds to $\mathcal{P}(P)$. However, this section shall consider arbitrary original lattices, in order to keep the results as general as possible.

Having fixed an original lattice, it is possible to restate the goal of this section in the following way: given an arbitrary original lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ and a security labelling $\Gamma : Var \to L$, the goal is to find the lowest closure operator Π over $\mathcal{L}$ such that the Smith *et al* type system types c with respect to Γ and $\Pi(L)$.

8.1 Confidentiality policies induced by security labellings

As noted in [17], "typically only a small subset of the subset lattice of a certain alphabet is used in applications". This happens, because establishing a given security labelling means selecting the security levels which will be really used. This section formalizes this argument and presents an illustrative example.

Lemma 33. *Given a lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ and a reduction operator Π over $\mathcal{L}$, $\Pi(L) = \{\Pi(l) \mid l \in L\}$ is a sublattice of the original lattice $\mathcal{L}$.*

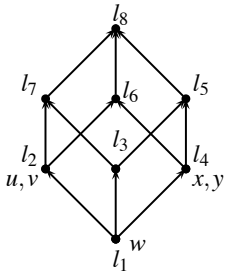
Proof. It suffices to prove that $\Pi(L)$ is closed to the upper bound in $\mathcal{L}$. Consider two arbitrary security levels $l_1, l_2 \in L$:

$\Pi(\Pi(l_1) \sqcup \Pi(l_2)) \sqsubseteq \Pi(l_1) \sqcup \Pi(l_2)$ (Π is reductive)
$\Pi(l_1) \sqsubseteq \Pi(l_1) \sqcup \Pi(l_2) \Rightarrow \Pi(l_1) \sqsubseteq \Pi(\Pi(l_1) \cup \Pi(l_2))$ (By monotonocity and extensivity)
$\Pi(l_2) \sqsubseteq \Pi(l_1) \sqcup \Pi(l_2) \Rightarrow \Pi(l_1) \sqsubseteq \Pi(\Pi(l_1) \cup \Pi(l_2))$ (By monotonocity and extensivity)
$\Pi(l_1) \sqcup \Pi(l_2) \sqsubseteq \Pi(\Pi(l_1) \cup \Pi(l_2))$

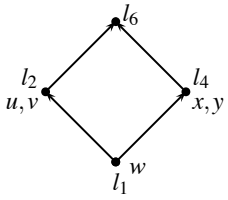
Thus, it follows that: $\Pi(\Pi(l_1) \cup \Pi(l_2)) = \Pi(l_1) \sqcup \Pi(l_2)$. Given two arbitrary security levels the greatest lower bound is given by: $\Pi(\Pi(l_1) \sqcap \Pi(l_2))$. One can easily check that the proposed lower bound is both closed for Π and a lower bound. To prove that it is indeed the greatest lower bound consider a security level $l_3 \in L$ such that $\Pi(l_3) = l_3$ and $l_3 \sqsubseteq \Pi(l_1) \sqcap \Pi(l_2)$, applying monotonocity and idempotence it follows that: $l_3 \sqsubseteq \Pi(\Pi(l_1) \sqcap \Pi(l_2))$. □

Lemma 24 states that $\alpha_\Gamma \circ \gamma_\Gamma$ is a reduction operator on $\mathcal{L}$ and lemma 33 recalls that the range of a reduction operator on a lattice is a sublattice of the original lattice. As such, when establishing a security labelling $\Gamma : Var \to L$, one is in fact establishing a reduction operator on $\mathcal{L}$ (henceforth denoted by Ξ_Γ) and thereby restricting the information flows that can take place to the lattice $\alpha_\Gamma \circ \gamma_\Gamma(L)$, which is illustrated in example 6.

Example 6. *This example illustrates a security labelling Γ which maps the set of variables $\{u, v, x, w, y\}$ to the lattice represented bellow. Each variable is depicted next to the security level to which it is mapped.*



The reduction operator that corresponds to this security labelling yields the following sublattice of the original lattice:



8.2 Computing the strictest confidentiality policy

Section 8.1 argues that when establishing a security labelling $\Gamma : Var \to L$, one is in fact establishing a reduction operator over $\mathcal{L}$ and thus selecting the possible information flows that can take place within any program which is executed under this security labelling. As such, the strictest security policy corresponds to a closure operator over $\Xi_\Gamma(L)$, rather than a closure operator over $\mathcal{L}$.

Making use of theorem 2, the goal of this section can be restated yet again as follows: given an original lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$, a security labelling $\Gamma : Var \to L$ and a program c, the goal is to find the smallest closure operator Π over $\Xi_\Gamma(L)$ such that:

$$\forall_{x \in Var} \,.\, \Pi(\Xi_\Gamma(\Gamma(x))) = \Pi(\bigsqcup\{\Xi_\Gamma(\Gamma(y)) \mid y \in \Delta(x)\})$$

Where $\Delta = \mathcal{D}[[c]](\varnothing, \Delta_0)$ and $\Delta_0 : Var \to \mathcal{P}(P)$ is the flow function which assings each variable to the singular set which contains it. It is important to emphasize that the upper bound which is present in the condition is in fact the original upper bound on $\mathcal{L}$ which coincides with the upper bound on $\Xi_\Gamma(L)$. Furhtermore, since $(\mathcal{P}(Var), \alpha_\Gamma, \gamma_\Gamma, \mathcal{L})$ is a Galois connection, it follows that for every variable $x \in Var$, $\Xi_\Gamma(\Gamma(x)) = \Gamma(x)$. As such the condition stated above can be restated as:

$$\forall_{x \in Var} \,.\, \Pi(\Gamma(x)) = \Pi(\bigsqcup\{\Gamma(y) \mid y \in \Delta(x)\}) = \Pi(\alpha_\Gamma(\Delta(x)))$$

Lemmas 34 and 35 establish an equivalent of this condition for security levels instead of variables, which allow computing the strictest closure operator that types the program in question.

Lemma 34. *Given a security lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$, a program c, a security labelling $\Gamma : Var \to L$ and a flow function Δ over $\mathcal{P}(Var)$, the least closure operator Π over L such that:*

$$\forall_{x \in Var} \,.\, \Pi(\Gamma(x)) = \Pi\left(\bigsqcup\{\Gamma(y) \mid y \in \Delta(x)\}\right) = \Pi(\alpha_\Gamma(\Delta(x)))$$

also verifies:

$$\forall_{X \subseteq Var} \,.\, \Pi(\alpha_\Gamma(X)) = \Pi(\alpha_\Gamma(\Delta^*(X)))$$

Proof. Recalling that lemma 24 guarantees that α_Γ is additive, it follows that:

$$\Pi(\alpha_\Gamma(X)) = \Pi\left(\bigsqcup_{x \in X} \alpha_\Gamma(x)\right)$$

As such, using monotonocity one can establish the following lower bound on $\Pi(\alpha_\Gamma(X))$:

$$\bigsqcup_{x \in X} \Pi(\alpha_\Gamma(x)) \sqsubseteq \Pi(\alpha_\Gamma(X))$$

Applying the hypothesis:

$$\bigsqcup_{x \in X} \Pi(\alpha_\Gamma(\Delta(x))) \sqsubseteq \Pi(\alpha_\Gamma(X))$$

Using monotonocity and idempotence:

$$\Pi\left(\bigsqcup_{x \in X} \alpha_\Gamma(\Delta(x))\right) \sqsubseteq \Pi(\alpha_\Gamma(X))$$

Recalling that both α_Γ and Δ^* are additive (lemmas 25 and 25) it follows that:

$$\Pi(\alpha_\Gamma(\Delta^*(X))) \sqsubseteq \Pi(\alpha_\Gamma(X))$$

Since the original goal is to obtain the smallest closure operator which verifies the first condition, one can just demand that: $\Pi(\alpha_\Gamma(\Delta^*(X))) = \Pi(\alpha_\Gamma(X))$. □

Lemma 35. *Given a security lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$, a program c, a security labelling $\Gamma : Var \to L$, a flow function Δ over $\mathcal{P}(Var)$, and a closure operator Π over L such that:*

$$\forall_{l \in L} \,.\, \Pi(l) = \Pi(\Xi_\Gamma(l)) = \Pi(\alpha_\Gamma(\gamma_\Gamma(l)))$$

Then, the following equivalence holds:

$$\forall_{l \in L} \,.\, \Pi(l) = \Pi(\alpha_\Gamma(\Delta^*(\gamma_\Gamma(l)))) \Leftrightarrow \forall_{X \subseteq Var} \,.\, \Pi(\alpha_\Gamma(X)) = \Pi(\alpha_\Gamma(\Delta^*(X)))$$

Proof.

- Direct Implication. Assume: $\forall_{l \in L} \,.\, \Pi(l) = \Pi(\alpha_\Gamma(\Delta^*(\gamma_\Gamma(l))))$.

– To prove: $\forall_{X \subseteq Var}\ .\ \Pi(\alpha_\Gamma(X)) \sqsubseteq \Pi(\alpha_\Gamma(\Delta^*(X)))$. Consider an arbitrary set $X \subseteq Var$. Since Δ^* and Π are both closure operators, they are both monotonic. Lemma 23 also guarantees that α_Γ is monotonic. Thus:

$$X \subseteq \Delta^*(X)$$
$$\Pi(\alpha_\Gamma(X)) \sqsubseteq \Pi(\alpha_\Gamma(\Delta^*(X)))$$

– To prove: $\forall_{X \subseteq Var}\ .\ \Pi(\alpha_\Gamma(\Delta^*(X))) \sqsubseteq \Pi(\alpha_\Gamma(X))$. Consider an arbitrary set $X \subseteq Var$. Considering the assumption, it follows that:

$$\Pi(\alpha_\Gamma(X)) = \Pi(\alpha_\Gamma(\Delta^*(\gamma_\Gamma(\alpha_\Gamma(X)))))$$

Lemma 24 ensures that $\gamma_\Gamma \circ \alpha_\Gamma$ is a closure operator, which means that $X \subseteq \gamma_\Gamma(\alpha_\Gamma(X))$. since Π, α_Γ and Δ^* are monotonic, their composition is also monotonic. Thus following:

$$\Pi(\alpha_\Gamma(\Delta^*(X))) \sqsubseteq \Pi(\alpha_\Gamma(\Delta^*(\gamma_\Gamma(\alpha_\Gamma(X))))) = \Pi(\alpha_\Gamma(X))$$

• Converse Implication. Assume $\forall_{X \subseteq Var}\ .\ \Pi(\alpha_\Gamma(X)) = \Pi(\alpha_\Gamma(\Delta^*(X)))$. Consider an arbitrary $l \in L$:

$$\Pi(l) = \Pi(\alpha_\Gamma(\gamma_\Gamma(l))) \quad (\Pi \text{ is a closure operator over } \Pi_\Gamma(L))$$
$$\Pi(l) = \Pi(\alpha_\Gamma(\Delta^*(\gamma_\Gamma(l)))) \quad (\text{Applying the assumption})$$

□

Theorem 4 guarantees that given a program c, an arbitrary security lattice $\mathcal{L}$ and a security labelling Γ, it is always possible to compute the smallest closure operator over $\mathcal{L}$, Π, such that c is typable with respect to $\Pi(\mathcal{L})$ and to the security mapping Γ. Furthermore, a constructive method for computing this closure operator is also provided.

Theorem 4. *Given a security lattice $\mathcal{L} = (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$, a program c, a security labelling $\Gamma : Var \to L$, and a closure operator Δ^* over $\mathcal{P}(Var)$ the smallest closure operator Π over $\Xi(L)$ such that:*

$$\forall_{l \in \Xi(L)}\ .\ \Pi(l) = \Pi(\alpha_\Gamma(\Delta^*(\gamma_\Gamma(l))))$$

is given by $\bigsqcup\{H^k(\alpha_\Gamma \circ \Delta^* \circ \gamma_\Gamma) \mid k \geq 0\}$, where $H : (\Xi(L) \to \Xi(L)) \to (\Xi(L) \to \Xi(L))$ is an operator on monotonic and extensive functions over $\Xi(L)$ defined in the following way:

$$H(\Pi) = \Pi \circ \alpha_\Gamma \circ \Delta^* \circ \gamma_\Gamma$$

Proof. First note that the set of extensive and monotonic operators over a given lattice is a lattice when equipped with pointwise subset inclusion. It suffices to note that it is closed to the usual upper bound. As such, condider any two extensive and monotonic operators Π_1 and Π_2:

- Suppose $l_1 \sqsubseteq l_2$. Since Π_1 and Π_2 are assumed to be monotonic, it follows that $\Pi_1(l_1) \sqsubseteq \Pi_1(l_2)$ and $\Pi_2(l_1) \sqsubseteq \Pi_2(l_2)$. Thus concluding that:

$$\Pi_1 \sqcup \Pi_2(l_1) \sqsubseteq \Pi_1 \sqcup \Pi_2(l_2)$$

- For any security level $l \in L$, $l \sqsubseteq \Pi_1(l)$ and $l \sqsubseteq \Pi_2(l)$, which means that $l \sqsubseteq \Pi_1 \sqcup \Pi_2(l)$.

H is closed for extensive and monotonic functions over $\Xi(L)$:

- $\forall_{l \in L} \,.\, H(\Pi)(l) = H(\Pi)(\Xi(l))$. Given a security level $l \in L$:

$$H(\Pi)(\Xi(l)) = \Pi(\alpha_\Gamma(\Delta^*(\gamma_\Gamma(\Xi(l))))) = \Pi(\alpha_\Gamma(\Delta^*(\gamma_\Gamma(\alpha_\Gamma(\gamma_\Gamma(l))))))$$

  Observing that $(\mathcal{P}(Var), \alpha_\Gamma, \gamma_\Gamma, \mathcal{L})$ is a Galois connection (lemma 23, it follows that $\gamma_\Gamma \circ \alpha_\Gamma \circ \gamma_\Gamma = \gamma_\Gamma$ and thus:

$$H(\Pi)(\Xi(l)) = H(\Pi)(l)$$

- $\forall_{l \in \Xi(L)} \,.\, H(\Pi)(l) = \Xi(H(\Pi)(l))$. Assume that $\forall_{l \in \Xi(L)} \,.\, \Pi(l) \in \Xi(L)$. Applying lemma 23:

$$\Xi(\alpha_\Gamma \circ \Delta^* \circ \gamma_\Gamma) = \alpha_\Gamma \circ \gamma_\Gamma \circ \alpha_\Gamma \circ \Delta^* \circ \gamma_\Gamma$$
$$\Xi(\alpha_\Gamma \circ \Delta^* \circ \gamma_\Gamma) = \alpha_\Gamma \circ \Delta^* \circ \gamma_\Gamma$$

  Thus, applying the assumption, it follows that $\forall_{l \in \Xi(L)} \,.\, H(\Pi)(l) \in \Xi(L)$.

- Consider a monotonic function over $\Xi(L)$, Π, since Π, α_Γ, Δ^* and γ_Γ are monotonic, their composition is also monotonic.

- Consider an extensive function over $\Xi(L)$, Π and a security level $l \in \Xi(l)$:

  $\alpha_\Gamma \circ \gamma_\Gamma(l) = l \Rightarrow l \sqsubseteq \alpha_\Gamma \circ \gamma_\Gamma(l)$
  $\gamma_\Gamma(l) \sqsubseteq \Delta^* \circ \gamma_\Gamma(l)$ (Since Δ^* is extensive)
  $\alpha_\Gamma \circ \gamma_\Gamma(l) \sqsubseteq \alpha_\Gamma \circ \Delta^* \circ \gamma_\Gamma(l)$ (Since α_Γ is monotone)
  $l \sqsubseteq \alpha_\Gamma \circ \Delta^* \circ \gamma_\Gamma(l) \sqsubseteq \Pi \circ \alpha_\Gamma \circ \Delta^* \circ \gamma_\Gamma(l)$ (Since Π is assumed extensive)

  It is important to emphasize that the extensivity claim does only refer to the elements of $\Xi(L)$.

Given any two functions over $\Xi(L)$, Π_1, Π_2, such that $\Pi_1 \sqsubseteq \Pi_2$, it follows immediately that $H(\Pi_1) \sqsubseteq H(\Pi_2)$. Thus concluding that H is monotonic. Clearly, any function which satisfies the condition of the lemma must be greater that $\alpha_\Gamma \circ \Delta^* \circ \gamma_\Gamma$. Since it is extensive and monotonic, it can be used as the initial function in the construction of the fixed point. As such, invoking Kleen's theorem, one can claim that the least fixed point of H does exist and can be computed as specified in the lemma. □

9 Related Work

Distributed flow policies Almeida-Matos [2] addresses the problem of dealing with distributed flow policies in a mobile code setting. At a technical level, the flow policies that are encoded in a program are captured as "declassification effects" in the form of flow policies that are deduced from a program by means of a primitive (and rather coarse) type system. The technical emphasis is on the dynamic usage of that information in controlling migration. Here we focus on the representation of flow policies aiming at optimality results. Flow policies can be seen as declassification policies, in the sense of the "acts for" relation of the Decentralized Label Model [16] and of Non-disclosure [3].

Dependencies Amtoft *et al.* introduce a Hoare logic to reason about input output independencies [4]. Ulteriorly, Hunt *et al.* introduce a family of flow sensitive type systems parametrizes in an arbitrary security lattice [13]. They remark that when the chosen security lattice correspond to $\mathcal{P}(Var)$, the corresponding flow sensitive type system is the De Morgan dual of Amtoft *et al.* independence logic. Again, flow sensitive type systems reason about input output dependencies and not trace dependencies as discussed in this report. Abadi *et al.* [1] propose a more general dependency calculus which aims to provide a common approach to several different analysis that arise in many different settings, such as: security, partial evaluation, program slicing and call-tracking.

Inference In the context of information flow, the type inference problem can be stated in the following way: when presented a partial security labeling, the problem consists in finding the lowest total security labeling without changing the specified labels [8]. Generally, this kind of inference mechanism extracts from the program a set of constraints which are ulteriorly passed to a constraint solver. The inference mechanism presented in Section 7 does not correspond to this problem, since it allows increasing the security level of the specified labels.

Combining static and dynamic analysis of information flow Le Guernic *et al.* [11] propose a combination of static and dynamic analysis, in the form of a semantics based monitor that communicates with an automaton that analyzes abstractions of program events at runtime. Sharing the idea of tracking dependencies encoded in a program in order to prevent information leaks during execution, Shroff *et al.*[20] use a runtime system that is augmented with a statically computed fixed point of dependencies.

10 Conclusion and Future Work

Management of complex security policies and integration with existing infrastructure have been pointed out [24] as important challenges for information flow security. This report addresses some significant obstacles, namely by relaxing the setting where "Language-based information-flow techniques require that the annotations in the program faithfully describe the desired policy":

- Desired security policies need not be formalized by means of annotations, as policies can be managed directly over the principals from which labels are built.
- Programs do not have to be fully annotated, as the weakest labelings can be inferred from partial annotations.
- An error reporting mechanism can assist the development process, by pointing out how existing annotations might conflict with the desired policy.

Furthermore, inter-operation between operating systems and information flow systems is eased:

- By working with security lattices that are constructed from sets of principals, which have concrete representation in usual operating systems.
- By offering a solution to the runtime problem of deciding access control permissions to executing programs.

As future work, we plan to generalize our results to more expressive languages and more sensitive information flow properties, in particular in order to handle programs in concurrent languages. Given that most of the theory is founded on syntactic principles, we believe that this extension should not pose a significant problem. We also envisage to use the strictest flow policy of a program as a means to deal with runtime updates [12] to the allowed flow policy where the program is running. A greater challenge on which we are currently working is to generalize our results in order to handle runtime principals [22]. Our long term aim is to study the applicability of our framework in increasing the precision and ease of use of existing implementations of programming languages with built-in information flow controls (namely JIF [15] and Flow Caml [21]).

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL*, pages 147–160. ACM Press, 1999.

[2] Ana Almeida Matos. Flow-policy awareness for distributed mobile code. In *Proceedings of CONCUR 2009 - Concurrency Theory*, volume to appear of *Lecture Notes in Computer Science*. Springer, 2009.

[3] Ana Almeida Matos and Gérard Boudol. On declassification and the non-disclosure policy. *Journal of Computer Security*, 17(5):549–597, 2009.

[4] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In *In SAS, 3148*, volume 3148 of *Lecture Notes in Computer Science*, pages 100–115. Springer-Verlag, 2004.

[5] K. J. Biba. Integrity considerations for secure computer systems. Technical report, The MITRE Corporation, 04 1977.

[6] N. Broberg and David Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Programming Languages and Systems. 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*. Springer Verlag, 2006.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[8] Zhenyue Deng and Geoffrey Smith. Type inference and informative error reporting for secure information flow. In *Proceedings of the 44th annual Southeast regional conference*, ACM-SE 44, pages 543–548, New York, NY, USA, 2006. ACM.

[9] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[10] Joseph A. Goguen and José Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.

[11] Gurvan Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *20th IEEE Computer Security Foundations Symposium*, pages 218–232, 2007.

[12] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic updating of information-flow policies. In *Workshop on Foundations of Comp. Security*, pages 7–18, 2005.

[13] Sebastian Hunt and David Sands. On flow-sensitive security types. In *POPL: 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 79–90. ACM Press, 2006.

[14] P. Li, Y. Mao, and S. Zdancewic. Information integrity policies. In *IN PROCEEDINGS OF THE WORKSHOP ON FORMAL ASPECTS IN SECURITY AND TRUST - FAST*, 2003.

[15] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241. ACM Press, 1999.

[16] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[17] Sergei A. Obiedkov, Derrick G. Kourie, and Jan H. P. Eloff. On lattices in access control models. In Henrik Schärfe, Pascal Hitzler, and Peter Øhrstrøm, editors, *Proceedings of the 14th International Conference on Conceptual Structures (ICCS 2006)*, volume 4068 of *Lecture Notes in Computer Science*, pages 374–387. Springer, 2006.

[18] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[19] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17:517–548, October 2009.

[20] Paritosh Shroff, Scott F. Smith, and Mark Thober. Securing information flow via dynamic capture of dependencies. *Journal of Computer Security*, 16:637–688, December 2008.

[21] V. Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), 2003.

[22] Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. *ACM Transactions on Programming Languages and Systems*, 30(1):6, 2007.

[23] Dennis M. Volpano, Geoffrey Smith, and Cynthia E. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2–3):167–188, 1996.

[24] S. Zdancewic. Challenges for information-flow security. In *1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*, 2004.

[25] Steve Zdancewic and Andrew Myers. Robust declassification. In *Computer Security Foundations Workshop, IEEE*, volume 0, page 15, Los Alamitos, CA, USA, 2001. IEEE Computer Society.