Secure Information Flow by Self-Composition

Gilles Barthe[1*] Pedro R. D'Argenio[2†] Tamara Rezk[1*]

[1]INRIA Sophia-Antipolis, Project EVEREST,
2004, Route de Lucioles, BP 93, 06902 Sophia-Antipolis Cedex, France
{Gilles.Barthe,Tamara.Rezk}@sophia.inria.fr

[2]Laboratoire d'Informatique Fondamentale, CMI, Université de Provence,
39, Rue Joliot Curie, 13453 Marseille Cedex 13, France
dargenio@cmi.univ-mrs.fr

Abstract

Non-interference is a high-level security property that guarantees the absence of illicit information leakages through executing programs. More precisely, non-interference for a program assumes a separation between secret inputs and public inputs on the one hand, and secret outputs and public outputs on the other hand, and requires that the value of public outputs does not depend on the value of secret inputs. A common means to enforce non-interference is to use an information flow type system. However, such type systems are inherently imprecise, and reject many secure programs, even for simple programming languages. The purpose of this paper is to investigate logical formulations of non-interference that allow a more precise analysis of programs. It appears that such formulations are often sound and complete, and also amenable to interactive or automated verification techniques, such as theorem-proving or model-checking.

We illustrate the applicability of our method in several scenarii, including a simple imperative language, a non-deterministic language, and finally a language with shared mutable data structures.

1. Introduction

Security models for mobile and embedded code often guarantee that downloaded applications will not perform a number of illegal operations, but also often fail to guarantee high-level security properties such as confidentiality, integrity, or availability. Thus it is important to develop mechanisms capable of enforcing these security properties.

Type systems provide a standard means of guaranteeing properties of programs; in particular, information flow type systems have been used to guarantee non-interference in a variety of contexts, see e.g. [1, 3, 5, 17, 30, 41, 43] and more particularly [35] for a recent survey of the field. While information flow type systems provide an attractive means to enforce non-interference, they often turn out to be overly conservative in practice. Indeed, many secure programs are rejected by information flow type systems. As illustrated by Joshi and Leino [22], this phenomenon already arises in the context of simple programming languages: take for example the program `x:=y; x:=0` where `y` is the secret variable not to be revealed, and `x` is the public variable which can be observed by the attacker: upon execution of the program, the final value of `x` is `0`, hence the attacker will not be able to guess the value of `y`. Yet such a program is rejected by a typical information flow type system [42, 41], on the ground that one program fragment taken in isolation is insecure. One remedy to this particular problem is to adopt a control-flow sensitive analysis [8]. However, information flow type systems and static analyses for non-interference are inherently imprecise, and reject too many secure programs for being widely used in practice. The situation is even further aggravated if the programming language includes some features that are notoriously difficult to handle precisely with type systems, such as aliasing or concurrency [38].

General-purpose logics such as Hoare logics or temporal logics provide a standard means to specify and verify functional and/or behavioral properties of pro-

∗ Partially supported by the ACI SPOPS and the IST Projects Inspired and Profundis.

† Supported by the European Community Project IST-2001-35304, AMETIST (http://ametist.cs.utwente.nl). On leave from FaMAF, Univ. Nacional de Córdoba, Argentina.

grams. More recently, several works have advocated their use to verify security properties of programs: in particular, Darvas, Hähnle and Sands [11] have recently shown how dynamic logic may be used to verify non-interference of (sequential) Java programs. One of their criteria is based on the observation that non-interference of a program P can be reduced to a property about a single program execution (universally quantified over all possible program inputs) of the program $P; P'$, where P' is a "renaming" of P.

The objective of this work is to build upon similar ideas to provide characterizations of non-interference in Hoare and temporal logics. Our characterizations, which apply to many languages and different notions of non-interference, are based on the idea of self-composition (that we rediscovered independently from [11]), and on the observation that Hoare logics and temporal logics are sound and often complete w.r.t. the programming language operational semantics. Further, they enable the use of existing general-purpose verification tools for verifying non-interference, and even to combine them with type systems for increasing automation in proofs.

In order to provide the reader with some intuition, let us first consider a simple deterministic imperative language featuring sequential composition and equipped with an evaluation relation $\langle P, \mu\rangle \Downarrow \nu$, where P is a program and μ, ν are memories, i.e. maps from the program variables of P to values. Further, assume that every memory μ is split into a public part $\mathsf{low}(\mu)$ and a private part $\mathsf{high}(\mu)$. With such a notation, termination-insensitive non-interference for P may be cast as for all memories μ, μ', ν, ν':

$$\begin{array}{c}[\langle P, \mu\rangle \Downarrow \nu \;\wedge\; \langle P, \mu'\rangle \Downarrow \nu' \;\wedge\; (\mathsf{low}\ \mu) = (\mathsf{low}\ \mu')] \\ \Rightarrow (\mathsf{low}\ \nu) = (\mathsf{low}\ \nu')\end{array}$$

Now let $\vec{x}$ be the low program variables of P, i.e. $\mathrm{dom}(\mu_L) = \mathrm{dom}(\nu_L) = \vec{x}$, and let $\vec{y}$ be the high program variables of P, i.e. $\mathrm{dom}(\mu_H) = \mathrm{dom}(\nu_H) = \vec{y}$, and let $[\vec{x}', \vec{y}'/\vec{x}, \vec{y}]$ be a renaming of the program variables of P with fresh variables, and let P' be the program $P[\vec{x}', \vec{y}'/\vec{x}, \vec{y}]$. Then, using $\uplus$ to denote the disjoint union of two memories, and ; to denote sequential composition, we have $\langle P, \mu\rangle \Downarrow \nu \;\wedge\; \langle P', \mu'\rangle \Downarrow \nu'$ iff $\langle P; P', \mu \uplus \mu'\rangle \Downarrow \nu_{\uplus}\nu'$. Hence we can recast non-interference as for all memories μ, μ', ν, ν':

$$\begin{array}{c}\langle P; P', \mu \uplus \mu'\rangle \Downarrow \nu \uplus \nu' \;\wedge\; (\mathsf{low}\ \mu) = (\mathsf{low}\ (\mu' \circ [\vec{x'}/\vec{x}])) \\ \Rightarrow (\mathsf{low}\ \nu) = (\mathsf{low}\ \nu' \circ [\vec{x'}/\vec{x}]))\end{array}$$

Next, we can use programming logics, which are sound and (relative) complete w.r.t. the operational semantic, to provide an alternative characterization of non-interference. If we use Hoare triples with a logic that is expressive enough to capture equality between memories, then non-interference can be characterized as:

$$\{\vec{x} = \vec{x'}\}P; P'\{\vec{x} = \vec{x'}\}$$

Let us now instantiate our characterisation to the program `x:=y; x:=0`. Taking $x \mapsto x'$ and $y \mapsto y'$ as the renaming function, the program is non-interferent iff

$$\{\mathtt{x} = \mathtt{x}'\}\ \mathtt{x} := \mathtt{y}; \mathtt{x} := 0; \mathtt{x}' := \mathtt{y}'; \mathtt{x}' := 0\ \{\mathtt{x} = \mathtt{x}'\}$$

which is easy to show using the rules of Hoare logics. More generally, the characterization provides us with a means to resort to existing verification tools to prove, or disprove, non-interference of a program.

Further, the characterization may be extended in several directions: first, it can be extended to any programming language that features an appropriate notion of "independent composition" operator, and that is equipped with an appropriate logic. We illustrate this point by considering a programming language with shared mutable data structures, and by using separation logic [31, 20] to provide a characterization of non-interference (see Section 7). Second, it can be extended to arbitrary relations between inputs and between outputs, as in e.g. [15]. This more general form of non-interference is useful for providing a characterization of some controlled forms of declassification, such as delimited information release, a form of declassification recently introduced by Sabelfeld and Myers [36].

While Hoare logics are adequate for characterizing termination insensitive notions of non-interference in a deterministic setting, we use temporal logic to characterize termination-sensitive non-interference for sequential languages and possibilistic non-interference for non-deterministic languages.

In summary, the main contribution of this paper is a detailed study of several logical frameworks for characterizing non-interference, both for sequential and concurrent non-deterministic programming languages. Our work extends and systematizes previous characterizations or criteria for non-interference based on general purpose logics, see Subsection 9.5, and allows us to conclude that such logics can be used in an appropriate fashion to provide a criterion for, or even to characterize non-interference (in the course of the paper, we pay special attention to the benefits of completeness, see in particular Subsection 9.1). Another minor contribution of our work is to provide methods to establish non-interference for languages for which no information flow type system is known, see in particular Section 7.

2. Preliminaries

Let Lang be the set of *programs* specifiable in a given programming language, with a distinguished program $\surd \in \mathsf{Lang}$ indicating successful termination, and let S, S', S_1, etc. range over Lang. Further, let Var be the set of *variables* which may appear in programs, and let x, x', x_1, y, z, etc. range over Var. We set $\mathsf{var}(S)$ to be the set of variable names appearing in the text of S, and for $y \notin \mathsf{var}(S)$, we define $S[y/x]$ to be the same program as S where all (free) occurrences of variable x are changed by variable y.

Assume given a set $\mathcal{M}$ be the set of all memories, and let μ, μ', etc. range over $\mathcal{M}$. Further, assume given a lookup function $\mathsf{v} : (\mathcal{M} \times \mathsf{Var}) \rightharpoonup \mathcal{V}$, where $\mathcal{V}$ is the set of values of the language under discussion, and let $\mathsf{var}(\mu)$ denote the set of all variables whose values is stored in μ. (This setting is compatible with dynamic object creation: for instance, if μ is a memory and x is a pointer to a list, $\mathsf{v}(\mu, x)$ returns the list represented by this pointer rather than its actual memory address value. In other words, we implicitly assume that all information stored in memories can only be accessed throughout variables.)

Our characterisations rely on the ability to separate a memory in two disjoint pieces of memories, and to update memories locally. Both operations are specified as follows. First, if $\mu_1, \mu_2 \in \mathcal{M}$ verify $\mathsf{var}(\mu_1) \cap \mathsf{var}(\mu_2) = \emptyset$, then there exists $\mu_1 \oplus \mu_2 \in \mathcal{M}$ such that if $x \in \mathsf{var}(\mu_1)$ then $\mathsf{v}(\mu_1 \oplus \mu_2, x) = \mathsf{v}(\mu_1, x)$, if $x \in \mathsf{var}(\mu_2)$ then $\mathsf{v}(\mu_1 \oplus \mu_2, x) = \mathsf{v}(\mu_2, x)$ and undefined otherwise. Notice that $\oplus$ is commutative. Second, if $\mu \in \mathcal{M}$, $x \in \mathsf{Var}$ and $d \in \mathcal{V}$, then $\mu[x \mapsto d] \in \mathcal{M}$ is a memory s.t. for all $y \in \mathsf{Var}$ $\mathsf{v}(\mu[x \mapsto d], y) = \mathbf{if}\ x = y\ \mathbf{then}\ d\ \mathbf{else}\ \mathsf{v}(\mu, y)$.

Example 1. Suppose a language which only manipulates integers, i.e. $\mathcal{V} = \mathbb{Z}$. Then $\mathcal{M}$ is the set of all functions $\mu : \mathsf{Var} \to \mathbb{Z}$ with $\mathsf{var}(\mu) = \mathrm{dom}(\mu)$, $\mathsf{v}(\mu, x) = \mu(x)$, $\oplus$ is the disjoint union of functions, and $\mu[x \mapsto d](y) = \mathbf{if}\ x = y\ \mathbf{then}\ d\ \mathbf{else}\ \mu(y)$.

The operational semantics of the programming language is given by the *transition system* $(\mathsf{Conf}, \rightsquigarrow)$ where $\mathsf{Conf} = \mathsf{Lang} \times \mathcal{M}$ is the set of *configurations* and $\rightsquigarrow \subseteq \mathsf{Conf} \times \mathsf{Conf}$ is the *transition relation*. We write $c \rightsquigarrow c'$ for $(c, c') \in \rightsquigarrow$ and $c \not\rightsquigarrow$ if there is no $c' \in \mathsf{Conf}$ such that $c \rightsquigarrow c'$. Further, we let $\rightsquigarrow^*$ denote the reflexive and transitive closure of $\rightsquigarrow$. Finally, we assume that $(\surd, \mu)$ indicates successful termination of the program with memory μ, and hence that for all $\mu \in \mathcal{M}$, $(\surd, \mu) \not\rightsquigarrow$. In contrast, we say that a configuration (S, μ) *does not terminate*, denoted by $(S, \mu)\bot$, if the execution of S on memory μ does not terminate (either because of an infinite execution or an abnormal stop as, e.g., deadlock), i.e., $\neg\exists \mu' : (S, \mu) \rightsquigarrow^* (\surd, \mu')$.

Example 2. The non-deterministic language Par is defined by

$$\begin{aligned} S ::= \quad & x := e \ \mid\ \mathsf{if}\ [\!]\ b_0 \to S_0\ \ldots [\!]\ b_n \to S_n\ \mathsf{fi} \\ & \mid\ S_1 \,;\, S_2\ \mid\ \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}\ \mid\ S_1 \mid\mid S_2 \end{aligned}$$

where e is an arithmetic expression and $b, b_0, \ldots, b_n$ are boolean expressions. The transition relation of Par is defined by the following rules (we omit symmetric rules for $S_1 \mid\mid S_2$), where memories are the functions of Example 1.

$$(x := e, \mu) \rightsquigarrow (\surd, \mu[x \mapsto \mu(e)])$$

$$\frac{(S_1, \mu) \rightsquigarrow (S_1', \mu')}{(S_1 \,;\, S_2, \mu) \rightsquigarrow (S_1' \,;\, S_2, \mu')} \qquad \frac{(S_1, \mu) \rightsquigarrow (\surd, \mu')}{(S_1 \,;\, S_2, \mu) \rightsquigarrow (S_2, \mu')}$$

$$\frac{(S_j, \mu) \rightsquigarrow (S_j', \mu') \qquad \mu(b_j)\ \text{holds}}{(\mathsf{if}\ [\!]\ b_0 \to S_0\ \ldots [\!]\ b_n \to S_n\ \mathsf{fi}, \mu) \rightsquigarrow (S_j', \mu')}\ 0 \le j \le n$$

$$\frac{(S, \mu) \rightsquigarrow (S', \mu') \qquad \mu(b)\ \text{holds}}{(\mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}, \mu) \rightsquigarrow (S' \,;\, \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}, \mu')}$$

$$\frac{\neg\mu(b)\ \text{holds}}{(\mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}, \mu) \rightsquigarrow (\surd, \mu)}$$

$$\frac{(S_1, \mu) \rightsquigarrow (S_1', \mu')}{(S_1 \mid\mid S_2, \mu) \rightsquigarrow (S_1' \mid\mid S_2, \mu')} \qquad \frac{(S_1, \mu) \rightsquigarrow (\surd, \mu')}{(S_1 \mid\mid S_2, \mu) \rightsquigarrow (S_2, \mu')}$$

We now turn to state three basic assumptions which force the operational semantics to enjoy some minimal restrictions that are essential for our results to hold. Assumptions 1 and 3 are seemingly obvious and satisfied by most of the languages (if not all). Nonetheless we need to make them explicit to set the ground of our general framework. Assumption 2 rules out some behaviour where memories are objects more complex than functions.

Assumption 1. *Transitions preserve the set of variables of a program. Moreover they do not affect the values of other variables than those appearing in the program.*

Notice that this assumption is not contradictory with object creation: a new object may be created but it can only be (directly or indirectly) referred through some variable in the text of the program.

Assumption 2. *Apart from its syntax, the semantics of a program depends only on the value of its own variables. Moreover, given a memory, it is always possible to find another one with the same values that can be separated in two parts, one of which contains exactly all information relevant to the program. (See Fact 1.6.)*

Assumption 2 imposes some restrictions on the memory manipulation. For example, if x is a pointer to a list and $\mathsf{v}(\mu, x)$ is considered to be the list represented by this pointer (rather than its actual address value), the address value cannot affect the control flow of a program. That is, for pointer variables x and y, $\mathsf{if}\ [\!]\ (x{=}y) \rightarrow S\ [\!]\ (x{\neq}y) \rightarrow S'\ \mathsf{fi}$ is not a valid program.

Assumption 3. *The operational semantics of the language* Lang *is independent of variable names.*

This assumption allows to change variable names without altering the program behaviour.

Assumptions 1, 2, and 3 can be formalised (see Appendix A) and from them Fact 1 below follows.

Fact 1 (After assumptions).

1. If $\mathsf{var}(S) = \mathsf{var}(\mu_1)$ and $(S, \mu_1 {\oplus} \mu_2) \rightsquigarrow^* (S', \mu')$ then there exists μ_1' such that $\mu' = \mu_1' {\oplus} \mu_2$.
2. If $\mathsf{var}(S) = \mathsf{var}(\mu_1)$ and $(S, \mu_1 {\oplus} \mu_2) \rightsquigarrow^* (S', \mu_1' {\oplus} \mu_2)$ then $(S, \mu_1 {\oplus} \mu_3) \rightsquigarrow^* (S', \mu_1' {\oplus} \mu_3)$ for any μ_3 (with $\mathsf{var}(\mu_2) = \mathsf{var}(\mu_3)$).
3. If $\mathsf{var}(S) = \mathsf{var}(\mu_1)$ and $(S, \mu_1 {\oplus} \mu_2)\bot$ then $(S, \mu_1 {\oplus} \mu_3)\bot$ for any μ_3 (with $\mathsf{var}(\mu_2) = \mathsf{var}(\mu_3)$).
4. If $y \notin \mathsf{var}(S)$ and $(S, \mu) \rightsquigarrow^* (S', \mu')$ then $(S[y/x], \mu[x \mapsto d][y \mapsto \mathsf{v}(\mu, x)]) \rightsquigarrow^* (S'[y/x], \mu'[x \mapsto d][y \mapsto \mathsf{v}(\mu', x)])$ for any value d.
5. If $y \notin \mathsf{var}(S)$ and $(S, \mu)\bot$ then $(S[y/x], \mu[x \mapsto d][y \mapsto \mathsf{v}(\mu, x)])\bot$ for any value d.
6. For every memory μ there are μ_1, μ_2 such that $\mathsf{var}(\mu_1) = \mathsf{var}(S)$ and $\forall x : \mathsf{v}(\mu, x) = \mathsf{v}(\mu_1 {\oplus} \mu_2, x)$.

It is not difficult to verify that Par satisfies the three assumptions above and complies to Fact 1.

3. Information Flow: Definitions

Let $\phi : \mathsf{Var} \rightarrow \mathsf{Var}$ be a partial injective function intended to relate *variables* of two programs. Let $\mathrm{dom}(\phi) = \{x_1, \ldots, x_n\}$[1] and let $\mathcal{I} \subseteq \mathcal{V}^n \times \mathcal{V}^n$ be a binary relation on tuples of values intended to determine the *indistinguishability criterion*. We say that memory μ is $(\phi, \mathcal{I})$-*indistinguishable* from μ', denoted by $\mu \sim_\phi^{\mathcal{I}} \mu'$, if $\langle(\mathsf{v}(\mu, x_1),.., \mathsf{v}(\mu, x_n)), (\mathsf{v}(\mu', \phi(x_1)),.., \mathsf{v}(\mu', \phi(x_n)))\rangle \in \mathcal{I}$.

Example 3. Let $L \subseteq \mathsf{Var}$ be the set of *low* (or *public*) variables of a program. Let $id_L : \mathsf{Var} \rightarrow \mathsf{Var}$ be the identity function on L and undefined otherwise. Then $\sim_{id_L}^{=}$ is the usual indistinguishability relation used to characterise non-interference. It relates memories whose public variables agree in their values meaning that these memories cannot be distinguished one from each other.

However, our definition of indistinguishability is more flexible. Let $H = \{p\}$ where p is a pointer to a list, and let avrg be the function that computes the average of a list, i.e. $\mathsf{avrg}([d_1, \ldots, d_n]) = \frac{d_1 + \cdots + d_n}{n}$. Then $\sim_{id_H}^{\mathcal{A}}$ cannot distinguish between memories μ and μ' which agree on the average value of the list to which p points, i.e. which verify $\mathsf{avrg}(\mathsf{v}(\mu, p)) = \mathsf{avrg}(\mathsf{v}(\mu, p))$.

Function ϕ is somehow redundant. It can always be encoded on $\mathcal{I}$. For instance, $\sim_{id_L}^{=}$ is equivalently defined by $\sim_{id}^{=_L}$, where id is the identity function and $=_L$ is the set $\{\langle(d_1, .., d_m, e_{m+1}, .., e_n), (d_1, .., d_m, e'_{m+1}, .., e'_n)\rangle \mid d_i, e_j, e'_j \in \mathcal{V}\}$, provided $L = \{x_1, .., x_m\}$. The need of ϕ will become evident in Section 4 when security is defined using composition and variable renaming.

The next proposition follows from definition of $\sim$.

Proposition 1. *For all* μ_1, μ_2, μ_1', μ_2', μ_1'', μ_2'', ϕ, *and* $\mathcal{I}$, *such that* $\phi : \mathsf{var}(\mu_1) \rightarrow \mathsf{var}(\mu_2)$, $\mu_1 {\oplus} \mu_1' \sim_\phi^{\mathcal{I}} \mu_2 {\oplus} \mu_2'$ *iff* $\mu_1 {\oplus} \mu_1'' \sim_\phi^{\mathcal{I}} \mu_2 {\oplus} \mu_2''$.

We now turn to the definitions of non-interference; unless otherwise specified, from now on we fix programs S_1 and S_2, functions $\phi, \phi' : \mathsf{var}(S_1) \rightarrow \mathsf{var}(S_2)$, and indistinguishability criteria $\mathcal{I}$ and $\mathcal{I}'$ which define relations $\sim_\phi^{\mathcal{I}}$ and $\sim_{\phi'}^{\mathcal{I}'}$.

$S_1 \approxeq_{\phi',\mathcal{I}'}^{\phi,\mathcal{I}} S_2$ (read "S_1 is termination sensitive non-interfering with S_2") if for any two input indistinguishable states, the successful execution of S_1 from one of these states, implies the successful execution of S_2 from the other state, with both executions ending in output indistinguishable states.

$S_1 \approx_{\phi',\mathcal{I}'}^{\phi,\mathcal{I}} S_2$ (read "S_1 is termination insensitive non interfering with program S_2") is a weaker concept in the sense that in addition S_2 is not required to terminate.

Finally, a program is (TS or TI) $(\mathcal{I}, \mathcal{I}')$-secure if, it is (TS or TI) non interfering with itself.

Definition 1.
1. $S_1 \approxeq_{\phi',\mathcal{I}'}^{\phi,\mathcal{I}} S_2$ if for all $\mu_1, \mu_2, \mu_1' \in \mathcal{M}$,

$$\big(\, \mu_1 {\sim_\phi^{\mathcal{I}}} \mu_2 \ \wedge\ (S_1, \mu_1) \rightsquigarrow^* (\surd, \mu_1') \,\big) \Rightarrow \exists \mu_2' {\in} \mathcal{M} : (S_2, \mu_2) \rightsquigarrow^* (\surd, \mu_2') \ \wedge\ \mu_1' {\sim_{\phi'}^{\mathcal{I}'}} \mu_2'.$$

2. $S_1 \approx_{\phi',\mathcal{I}'}^{\phi,\mathcal{I}} S_2$ if for all $\mu_1, \mu_2, \mu_1' \in \mathcal{M}$,

$$\big(\, \mu_1 {\sim_\phi^{\mathcal{I}}} \mu_2 \ \wedge\ (S_1, \mu_1) \rightsquigarrow^* (\surd, \mu_1') \,\big) \Rightarrow \big(\, (S_2, \mu_2)\bot \ \vee\ \exists \mu_2' {\in} \mathcal{M} : (S_2, \mu_2) \rightsquigarrow^* (\surd, \mu_2') \ \wedge\ \mu_1' {\sim_{\phi'}^{\mathcal{I}'}} \mu_2' \,\big).$$

3. Let $\mathcal{I}, \mathcal{I}' \subseteq \mathcal{V}^n \times \mathcal{V}^n$ with $n = \#\,\mathsf{var}(S)$.
   (a) S is *termination sensitive (TS)* $(\mathcal{I}, \mathcal{I}')$*-secure* iff $S \approxeq_{id,\mathcal{I}'}^{id,\mathcal{I}} S$.
   (b) S is *termination insensitive (TI)* $(\mathcal{I}, \mathcal{I}')$*-secure* iff $S \approx_{id,\mathcal{I}'}^{id,\mathcal{I}} S$.

Traditional non-interference is characterised in our setting by $(=_L, =_L)$-security, whit $=_L$ as defined above. It is not difficult to check that our definitions agree with

---

1 We suppose variables can always be arranged in a particular order which we use to arrange set of variables in tuples.

those already defined in the literature (e.g. [16, 42, 38, 22]).

However, our definitions are in fact more flexible that the usual formulations of non-interference. Indeed, the latter usually require that executions from indistinguishable states ends at indistinguishable states with identical criteria of indistinguishability. In contrast, we allow indistinguishability for initial states (input indistinguishability) to differ differ from indistinguishability for final states (output indistinguishability). More precisely, Definition 1 identifies input indistinguishability with $(\phi, \mathcal{I})$-indistinguishability and output indistinguishability with $(\phi', \mathcal{I}')-$indistinguishability.

Furthermore, the generality of our definition is useful for providing a characterisation of some forms of controlled declassification. Declassification allows to leak some confidential information without being too revealing. A typical example is a program S that informs the average salary of the employees of a company without revealing any other information that may give any further indications of particular salaries (which is confidential information). Suppose salaries are stored in the list *lsalaries* and the average will be stored in a_l. S should be $(\mathcal{A}, =_{\{a_l\}})$-secure to ensure that no other information of the salary than the average is revealed. (See Example 3 for the definition of $\mathcal{A}$.) A semantic characterisation of this kind of properties has recently been given in [36] and coined *delimited release.*

4. Checking Information Flow using Composition and Renaming

Let $\triangleright$ be an operation in Lang such that, for all S_1, S_2, μ_1, μ_2, μ_1', μ_2', with $\mathsf{var}(S_1) \cap \mathsf{var}(S_2) = \emptyset$, $\mathsf{var}(S_1) = \mathsf{var}(\mu_1)$, $\mathsf{var}(S_2) = \mathsf{var}(\mu_2)$

(a) $(S_1, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2)$ iff $(S_1 \triangleright S_2, \mu_1 \oplus \mu_2) \rightsquigarrow^* (S_2, \mu_1' \oplus \mu_2)$; and

(b) $(S_1, \mu_1 \oplus \mu) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu)$ and $(S_2, \mu' \oplus \mu_2) \rightsquigarrow^* (\surd, \mu' \oplus \mu_2')$, for some μ and μ', iff $(S_1 \triangleright S_2, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2')$.

It is not difficult to check that sequential composition and parallel composition in language Par satisfy conditions of $\triangleright$.

Operation $\triangleright$ is the first of the two ingredients on which our result builds up. Notice that non-interference, as given in Definition 1, considers separately an execution of program S_1 and another of S_2. By composing $S_1 \triangleright S_2$, properties (a) and (b) above allows to put these executions one after the other. Therefore we can find a different characterisation of security:

Definition 2. Let S_1 and S_2 be two programs such that $\mathsf{var}(S_1) \cap \mathsf{var}(S_2) = \emptyset$. $S_1 \overset{\triangleright}{\approx}{}^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$ (resp. $S_1 \overset{\triangleright}{\sim}{}^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$) if for all μ_1, μ_2, μ_1', with $\mathsf{var}(\mu_1) = \mathsf{var}(\mu_1') = \mathsf{var}(S_1)$ and $\mathsf{var}(\mu_2) = \mathsf{var}(S_2)$,

$$\begin{aligned}
&(\, \mu_1 \oplus \mu_2 \sim^{\mathcal{I}}_{\phi} \mu_1 \oplus \mu_2 \;\wedge\; (S_1 \triangleright S_2, \mu_1 \oplus \mu_2) \rightsquigarrow^* (S_2, \mu_1' \oplus \mu_2) \,) \Rightarrow \\
&\quad \exists \mu_2' : \mathsf{var}(\mu_2') = \mathsf{var}(S_2) : \\
&\qquad (S_2, \mu_1' \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \;\wedge\; \mu_1' \oplus \mu_2' \sim^{\mathcal{I}'}_{\phi'} \mu_1' \oplus \mu_2' \\
&\quad (\vee\ (S_2, \mu_1' \oplus \mu_2)\bot \text{ for the TI case}).
\end{aligned}$$

Notice that this definition has the same shape as Definition 1. However, while $S_1 \approx^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$ considers executions of two different programs (S_1 and S_2), $S_1 \overset{\triangleright}{\approx}{}^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$ considers the execution of only one program ($S_1 \triangleright S_2$): first, the execution until the middle (that is, until S_2) and then the execution until the end.

The next theorem claims that Definitions 1 and 2 are equivalent. That is, non-interference of two programs can be seen as non-interference of one program (namely, the composition of those two programs). The proof can be found in Appendix B.

Theorem 1. *Let S_1 and S_2 such that $\mathsf{var}(S_1) \cap \mathsf{var}(S_2) = \emptyset$. Then*
(a) $S_1 \approx^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$ if and only if $S_1 \overset{\triangleright}{\approx}{}^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$, and
(b) $S_1 \approx^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$ if and only if $S_1 \overset{\triangleright}{\sim}{}^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$.

Programs sharing variable names fall out of Definition 2 (and Theorem 1). Using variable renaming — the second ingredient— conflicting variables can be renamed to fresh names and hence the definition can be adapted to a more general setting. For this, we need to ensure that the behaviour of the renamed program is the same (which is guaranteed by Assumption 3), and that non-interference is preserved by renaming, which is stated in the following theorem.

Theorem 2. *Let $\xi : \mathsf{var}(S_2) \to V$ be a bijective function on a set of variables V. Then*
(a) $S_1 \approx^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$ iff $S_1 \approx^{\xi\circ\phi,\mathcal{I}}_{\xi\circ\phi',\mathcal{I}'} S_2[\xi]$, and
(b) $S_1 \approx^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$ iff $S_1 \approx^{\xi\circ\phi,\mathcal{I}}_{\xi\circ\phi',\mathcal{I}'} S_2[\xi]$.
where $S_2[\xi]$ is program S_2 whose variables has been renamed according to function ξ.

Putting together Theorems 1 and 2 we have the following corollary:

Corollary 1. *Let $\mathsf{var}(S)' = \{x' \mid x \in \mathsf{var}(S)\}$ such that $\mathsf{var}(S) \cap \mathsf{var}(S)' = \emptyset$. Let $\xi : \mathsf{var}(S) \to \mathsf{var}(S)'$ such that $\xi(x) = x'$ for all $x \in \mathsf{var}(S)$. Then, the following statements are equivalent*

1. *S is TS (resp. TI) $(\mathcal{I}_1, \mathcal{I}_2)$-secure.*
2. *$S \approx^{\xi,\mathcal{I}_1}_{\xi,\mathcal{I}_2} S[\xi]$ (resp. $S \approx^{\xi,\mathcal{I}_1}_{\xi,\mathcal{I}_2} S[\xi]$)*
3. *$S \overset{\triangleright}{\approx}{}^{\xi,\mathcal{I}_1}_{\xi,\mathcal{I}_2} S[\xi]$ (resp. $S \overset{\triangleright}{\sim}{}^{\xi,\mathcal{I}_1}_{\xi,\mathcal{I}_2} S[\xi]$)*

Corollary 1 allows to analyze whether a program S is secure in a single execution of the program $S \triangleright S[\xi]$. But this is what verification logic are used to. After revisiting Definition 2 in a deterministic setting, we characterise $(\mathcal{I}_1, \mathcal{I}_2)$-security in some such logics.

5. Deterministic Programs

Simpler definitions for non interference can be obtained if the program S under study is deterministic as stated by the following theorem.

Theorem 3. *Let $S \triangleright S[\xi]$ be a deterministic program (and hence also is S), and let ξ as in Corollary 1.*
1. S is TS $(\mathcal{I}_1, \mathcal{I}_2)$-secure if and only if

$$\begin{aligned}
&\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge\ \mathsf{var}(\mu_2) = \mathsf{var}(S)' : \\
&\mu_1 \oplus \mu_2 \sim_\xi^{\mathcal{I}_1} \mu_1 \oplus \mu_2 \\
&\Rightarrow (S, \mu_1 \oplus \mu_2) \bot \ \vee \\
&\quad \exists \mu_1', \mu_2' : \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge\ \mathsf{var}(\mu_2') = \mathsf{var}(S)' : \\
&\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge\ \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'
\end{aligned}$$

2. S is TI $(\mathcal{I}_1, \mathcal{I}_2)$-secure if and only if

$$\begin{aligned}
&\forall \mu_1, \mu_2, \mu_1', \mu_2' : \\
&\mathsf{var}(\mu_1) = \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge\ \mathsf{var}(\mu_2) = \mathsf{var}(\mu_2') = \mathsf{var}(S)' : \\
&(\mu_1 \oplus \mu_2 \sim_\xi^{\mathcal{I}_1} \mu_1 \oplus \mu_2 \ \wedge\ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2')) \\
&\Rightarrow \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'
\end{aligned}$$

The simplicity of definitions in Theorem 3 w.r.t. Definition 2 lies in the fact that we do not need to make reference to intermediate points in the program. This allows to check security by simply analysing the I/O behaviour of program $S; S[\xi]$.

The theorem can be further enhanced for languages featuring simple functional memories like the one defined in Example 1. Notice that Theorem 3 requires that memory should be "separable" by operation $\oplus$. Functions can *always* be separated. Consequently, we have the following corollary:

Corollary 2. *Let $S \triangleright S[\xi]$ be a deterministic program with memory as defined in Example 1. Let ξ as in Corollary 1.*
1. S is TS $(\mathcal{I}_1, \mathcal{I}_2)$-secure if and only if

$$\begin{aligned}
&\forall \mu : \mu \sim_\xi^{\mathcal{I}_1} \mu \Rightarrow \\
&\quad (\, (S, \mu) \bot \ \vee\ \exists \mu' : (S \triangleright S[\xi], \mu) \rightsquigarrow^* (\surd, \mu') \ \wedge\ \mu' \sim_\xi^{\mathcal{I}_2} \mu' \,)
\end{aligned}$$

2. S is TI $(\mathcal{I}_1, \mathcal{I}_2)$-secure if and only if

$$\forall \mu, \mu' : (\, \mu \sim_\xi^{\mathcal{I}_1} \mu \ \wedge\ (S \triangleright S[\xi], \mu) \rightsquigarrow^* (\surd, \mu') \,) \Rightarrow \mu' \sim_\xi^{\mathcal{I}_2} \mu'$$

6. Hoare logic

Let While be the subset of Par not containing parallel composition and limiting the if construction to be binary and deterministic: $\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}$ = $\mathsf{if}\ []\ b \to S_1\ []\ \neg b \to S_2\ \mathsf{fi}$. Recall that memories are the functions of Example 1.

Let P and Q be first order predicate and S a While program. Recall that a Hoare triple [19] $\{P\}\, S\, \{Q\}$ means that whenever S starts to execute in a state in which P holds, if it terminates, it does so in a state satisfying Q. An assertion $\{P\}\, S\, \{Q\}$ holds if it is provable with the following rules:

$$\{P[e/x]\}\, x := e\, \{P\} \qquad \frac{P' \to P \quad \{P\}\, S\, \{Q\} \quad Q \to Q'}{\{P'\}\, S\, \{Q'\}}$$

$$\frac{\{P \wedge b\}\, S_1\, \{Q\} \quad \{P \wedge \neg b\}\, S_2\, \{Q\}}{\{P\}\, \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}\, \{Q\}}$$

$$\frac{\{P\}\, S_1\, \{R\} \quad \{R\}\, S_2\, \{Q\}}{\{P\}\, S_1\, ;\, S_2\, \{Q\}} \qquad \frac{\{P \wedge b\}\, S\, \{P\}}{\{P\}\, \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}\, \{P \wedge \neg b\}}$$

Hoare logic is sound and (relatively) complete w.r.t. operational semantics [10]. That is, for all program S and predicates P and Q, $\{P\}\, S\, \{Q\}$ is provable iff for all μ, μ', $\mu \models P$ and $(S, \mu) \rightsquigarrow^* (\surd, \mu')$ imply $\mu' \models Q$. $\mu \models P$ means that P holds whenever every program variable x appearing in P is replaced by the value $\mathsf{v}(\mu, x)$.

Suppose $\mathbf{I}(\mathcal{I})$ is a predicate such that

$$\mu \models \mathbf{I}(\mathcal{I}) \quad \text{iff} \quad \mu \sim_\xi^{\mathcal{I}} \mu \quad \big(\text{iff} \quad (\mathsf{v}(\mu, \vec{x}), \mathsf{v}(\mu, \vec{x}')) \in \mathcal{I}\big).$$

where $\mathsf{v}(\mu, (x_1, .., x_n)) = (\mathsf{v}(\mu, x_1), .., \mathsf{v}(\mu, x_n))$ and $\mathsf{var}(S) = \{x_1, .., x_n\}$. We expect that $\mathbf{I}(\mathcal{I})$ is definable in the specification logic embedded in the Hoare logic. For instance, predicate $\mathbf{I}(=_L)$ for relation $\sim_\xi^{=_L}$ (which is the renaming version of $\sim_{id_L}^{=}$ in Example 3), can be defined by $\bigwedge_{x \in L} x = x'$.

Termination insensitive $(\mathcal{I}_1, \mathcal{I}_2)$-security can be characterised in Hoare logic as follows: S is TI $(\mathcal{I}_1, \mathcal{I}_2)$-secure iff $\{\mathbf{I}(\mathcal{I}_1)\}\, S\, ;\, S[\xi]\, \{\mathbf{I}(\mathcal{I}_2)\}$ is provable. This is shown in the following:

S is TI $(\mathcal{I}_1, \mathcal{I}_2)$-secure
iff {Corollary 2.2}
$\forall \mu, \mu' : (\, \mu \sim_\xi^{\mathcal{I}_1} \mu \ \wedge\ (S\, ;\, S[\xi], \mu) \rightsquigarrow^* (\surd, \mu') \,) \Rightarrow \mu' \sim_\xi^{\mathcal{I}_2} \mu'$
iff {Def. of $\mathbf{I}$}
$\forall \mu, \mu'$:
$(\, \mu \models \mathbf{I}(\mathcal{I}_1) \ \wedge\ (S\, ;\, S[\xi], \mu) \rightsquigarrow^* (\surd, \mu') \,) \Rightarrow \mu' \models \mathbf{I}(\mathcal{I}_2)$
iff {Soundness and completeness, provided $\mathbf{I}$ is definable}
$\{\mathbf{I}(\mathcal{I}_1)\}\, S\, ;\, S[\xi]\, \{\mathbf{I}(\mathcal{I}_2)\}$ is provable

Example 4. Let x_l and y_h be respectively a public and a confidential variable in the program $x_l := x_l + y_h\, ;\, x_l := x_l - y_h$. We show that it is noninterfering. Indistinguishability in this case is characterised by predicate $\mathbf{I}(=_{\{x_l\}}) \equiv (x_l = x_l')$. The proof is given in the left side of Figure 1.

Another example —the PIN access control— deals with declassification. In the program

$$\mathsf{if}\ (in = pin)\ \mathsf{then}\ acc := \mathsf{true}\ \mathsf{else}\ acc := \mathsf{false}\ \mathsf{fi}$$

$$
\begin{array}{ll}
\{x_l = x'_l\} & \{(in = pin) \leftrightarrow (in' = pin')\} \\
\{x_l + y_h - y_h = x'_l\} & \quad \mathsf{if}\ (in = pin)\ \mathsf{then} \\
\quad x_l := x_l + y_h\ ; & \quad\quad \{in' = pin'\} \\
\{x_l - y_h = x'_l\} & \quad\quad acc := \mathsf{true} \\
\quad x_l := x_l - y_h\ ; & \quad \mathsf{else} \\
\{x_l = x'_l\} & \quad\quad \{in' \neq pin'\} \\
\{x_l = x'_l + y'_h - y'_h\} & \quad\quad acc := \mathsf{false} \\
\quad x'_l := x'_l + y'_h\ ; & \quad \mathsf{fi}\ ; \\
\{x_l = x'_l - y'_h\} & \{(in' = pin') \leftrightarrow (acc = \mathsf{true})\} \\
\quad x'_l := x'_l - y'_h & \quad \mathsf{if}\ (in' = pin')\ \mathsf{then}\ acc' := \mathsf{true} \\
\{x_l = x'_l\} & \quad\quad\quad\quad \mathsf{else}\ acc' := \mathsf{false}\ \mathsf{fi} \\
 & \{acc = acc'\}
\end{array}
$$

Figure 1. Security proof in Hoare logic

variable pin, which stores the actual PIN number, is supposed to be confidential, whereas in, containing the attempted number, is a public input variable and acc, conceding or not the access to the system, is a public output variable. The declassified information only should reveal whether the input number (in) agrees with the PIN number (pin) or not, and such information is revealed by granting the access or not (indicated in acc). We, therefore, require that the program is $(\mathcal{I}, =_{\{acc\}})$-secure, where $\mathcal{I}$ is such that $\sim^{\mathcal{I}}_{id}$ iff $(\mu(in) = \mu(pin)) \Leftrightarrow (\mu'(in) = \mu'(pin))$. Hence, $\mathbf{I}(\mathcal{I}) \equiv ((in = pin) \leftrightarrow (in' = pin'))$ and $\mathbf{I}(=_{\{acc\}}) \equiv (acc = acc')$. The proof is suggested in the right side of Figure 1.

7. Separation Logic

Separation logic is an extension of Hoare logic to reason about shared mutable data structures [31, 20]. While^p extends the While language with the following commands:

$$S ::= \ x := e.i \mid x.i := e \mid x := \mathsf{cons}(e_1, e_2) \mid \cdots \qquad (1)$$

where $i \in \{1, 2\}$ and e is a pure expression (not containing a dot or cons). $x := \mathsf{cons}(e_1, e_2)$ creates a cell in the heap where the tuple (e_1, e_2) is stored, and allows x to point to that cell. $e.i$ returns the value of the ith position of the tuple pointed by e. (Binary tuples suffice for our purposes although arbitrary n-tuples appear in the literature and can also be considered here.) Then, $x := e.i$ and $x.i := e$ allow to read and update the heap respectively. Values in While^p may be integers or locations (including nil).

A memory contains two components: a store, mapping variables into values, and a heap, mapping locations (or addresses) into values. Thus, if $\mathcal{V} = \mathbb{Z} \cup \mathsf{Loc}$, $\mathcal{S} = \mathsf{Var} \to \mathcal{V}$ is the set of *stores* and $\mathcal{H} = \mathsf{Loc} - \{\mathsf{nil}\} \to (\mathcal{V} \times \mathcal{V})$ is the set of *heaps*. Hence $\mathcal{M} = \mathcal{S} \times \mathcal{H}$. As a consequence variables can have type $\mathbb{Z}$ or type Loc.

Separation logic requires additional predicates to manipulate pointers. In addition to formulas of the classical predicate calculus, the logic has the following forms of assertions: $e \mapsto (e_1, e_2)$ that holds in a singleton heap with location satisfying e and the cell values satisfying e_1 and e_2 respectively; empty that holds if the heap is empty; and $P * Q$, named *separating conjunction*, holds if the heap can be split in two parts, one satisfying P and the other Q. There exists a calculus for these operations including the separating implication $P \mathrel{-\!\!*} Q$, see [20, 31]. The meaning of an assertion depends upon both the store and the heap:

$$
\begin{array}{lll}
(s, h) \models \mathsf{empty} & \text{iff} & \mathrm{dom}(h) = \emptyset \\
(s, h) \models e \mapsto (e_1, e_2) & \text{iff} & \mathrm{dom}(h) = \{s(e)\} \text{ and} \\
 & & \quad h(s(e)) = (s(e_1), s(e_2)) \\
(s, h) \models P * Q & \text{iff} & \exists h_0, h_1 : h_0 \oplus h_1 = h, \\
 & & \quad (s, h_0) \models P, \text{ and } (s, h_1) \models Q
\end{array}
$$

Separation logic extends Hoare logic with rules to handle pointers. The so-called *frame rule*, that allows to extend *local* specification, is given by

$$\frac{\{P\}\,S\,\{Q\}}{\{P * R\}\,S\,\{Q * R\}}$$

where no variable occurring free in R is modified by S. The (local version) rules for heap manipulation commands are the following (we omit symmetric rules):

Let $e \mapsto (\_, e_2)$ abbreviates "$\exists e' : e \mapsto (e', e_2)$ and e' is not free in e", then

$$\{e \mapsto (\_, e_2)\}\, e.i := e_1\, \{e \mapsto (e_1, e_2)\}$$

If x does not occur in e_1 or in e_2 then

$$\{\mathsf{empty}\}\, x := \mathsf{cons}(e_1, e_2)\, \{x \mapsto (e_1, e_2)\}$$

If x, x' and x'' are different and x does not occur in e, then

$$\big\{x{=}x' \wedge (e \mapsto (x'', e_2))\big\}\, x := e.1\, \big\{x{=}x'' \wedge (e \mapsto (x'', e_2))\big\}$$

Using separation logic we can define inductive predicates to make reference to structures in the heap (see [31, 20]). For simplicity we only consider predicate list which is defined by:

$$
\begin{array}{rcl}
\mathsf{list}.[\,].p & = & (p = \mathsf{nil}) \\
\mathsf{list}.(x{:}xs).p & = & (\exists r : (p \mapsto (x, r)) * \mathsf{list}.xs.r)
\end{array}
$$

As we mentioned, a memory is a tuple containing a store and a heap. We need to define var, v, and $\oplus$ in this domain. Therefore, for all $s, s_1, s_2 \in \mathcal{S}$, $h, h_1, h_2 \in \mathcal{H}$, and $x \in \mathsf{Var}$, we define $\mathsf{var}(s, h) = \mathrm{dom}(s)$,

$$\mathsf{v}((s, h), x) \ = \ \begin{cases} s(x) & \textbf{if } s(x) \in \mathbb{Z} \\ \overline{\mathsf{v}}(h, s(x)) & \textbf{if } s(x) \in \mathsf{Loc} \end{cases}$$

where $\overline{\mathsf{v}}(h,l)$ returns the list pointed by l, i.e., $\overline{\mathsf{v}}(h,l) =$ **if** l=nil **then** [ ] **else** $\mathrm{fst}(h(l))$:$\overline{\mathsf{v}}(h-\{(l,h(l))\},\ \mathrm{snd}(h(l)))$; and

$$(s_1,h_1)\oplus(s_2,h_2) \ = \ (s_1\oplus s_2, h_1\oplus h_2)$$

defined only if $\mathrm{reach}(\mathrm{ran}(s_i)\cap\mathsf{Loc}-\{\mathsf{nil}\},h_i) \subseteq \mathrm{dom}(h_i)$ for all $i=1,2$, where $\mathrm{reach}(loc,h) = \bigcup_{n\geq 0} h^n(loc) - \{\mathsf{nil}\}$ is the set of locations reachable form the set loc. If this restriction does not hold, then, for $x \in \mathsf{var}(s_1,h_1)$, $\mathsf{v}((s_1\oplus s_2, h_1\oplus h_2),x)$ may be defined when $\mathsf{v}((s_1,h_1),x)$ is not (hence not satisfying the requirement of $\oplus$ in Section 2).

Notice that $(s,h) \models \mathsf{list}.xs.x$ iff $\overline{\mathsf{v}}(h,s(x)) = xs$ iff $\mathsf{v}((s,h),x) = xs$.

Let $\{x_1,..,x_n\} = \mathsf{var}(S)\cap\mathsf{Loc}$ and $\{y_1,..,y_m\} = \mathsf{var}(S)-\mathsf{Loc}$. Let $\vec{x} = (x_1,..,x_n)$ and $\vec{x}' = (x_1',..,x_n')$ and similarly for $\vec{y}$ and $\vec{y}'$. Denote $\vec{xs} = (xs_1,..,xs_1)$ and $\vec{xs}' = (xs_1',..,xs_n')$. Fix this notation for the rest of this section. Let $\mathbf{I}_{sl}(\mathcal{I})$ be predicate

$$\exists \vec{xs},\vec{xs}' : \Big(\Big(\bigwedge_{1\leq i\leq n} \mathsf{list}.xs_i.x_i\Big) * \Big(\bigwedge_{1\leq i\leq n} \mathsf{list}.xs_i'.x_i'\Big) \Big) \wedge \mathbf{I}_v(\vec{xs},\vec{xs}',\mathcal{I})$$

where we suppose the existence of $\mathbf{I}_v$ such that

$$\mu \models \mathbf{I}_v(\vec{ds},\vec{ds}',\mathcal{I}) \quad \text{iff} \quad (\langle \mathsf{v}(\mu,\vec{y}),\vec{ds}\rangle, \langle \mathsf{v}(\mu,\vec{y}'),\vec{ds}'\rangle) \in \mathcal{I}$$

where $\mathsf{v}(\mu,\vec{y})$ is defined as in Section 6, and $\vec{ds}$ and $\vec{ds}'$ are actual list values. (Notice that $\mu \models \mathbf{I}_v(\mathsf{v}(\mu,\vec{x}),\mathsf{v}(\mu,\vec{x}'),\mathcal{I})$ iff $(\langle \mathsf{v}(\mu,\vec{y}),\mathsf{v}(\mu,\vec{x})\rangle, \langle \mathsf{v}(\mu,\vec{y}'),\mathsf{v}(\mu,\vec{x}')\rangle) \in \mathcal{I}$ iff $\mu \sim^{\mathcal{I}}_{\xi} \mu$.)

$\mathbf{I}_{sl}(\mathcal{I})$ has two parts: the first part states the separation of the heap, the second one, the indistinguishability of the values.

Separation logic is (relatively) complete for the sublanguage we are using [20] (recall that it restricts to use the dot and cons only in the specific assignments in (1)). As a consequence, security in separation logic can be completely characterised as follows: S is TI $(\mathcal{I}_1,\mathcal{I}_2)$-secure iff $\{\mathbf{I}_{sl}(\mathcal{I}_1)\}\, S\,;S[\xi]\, \{\mathbf{I}_{sl}(\mathcal{I}_2)\}$ is provable. This can be proven like for Hoare logic using Theorem 3 instead of Corollary 2, and the characterisation of $\mathbf{I}_{sl}(\mathcal{I})$ above. The details are reported in Appendix C.

Example 5. The following program receives a list *lsalaries* with employees salaries and returns in a_l the average of the salaries.

```
p := lsalaries ; s := 0 ; n := 0 ;
while p ≠ nil do
   n := n + 1 ; s_aux := p.salary ; s := s + s_aux ;
   p_aux := p.next ; p := p_aux ;
od
a_l := s/n
```

Variables s_{aux} and p_{aux} are specially included to meet the syntax restrictions imposed to the language. Call this program $\mathsf{AV\_SAL}$ (for "AVerage SALary").

The security requirement have been discussed at the end of Section 3: we would like it to be $(\mathcal{A},=_{\{a_l\}})$-secure. Thus, precondition $\mathbf{I}_{sl}(\mathcal{A})$ and postcondition $\mathbf{I}_{sl}(=_{\{a_l\}})$ are respectively the following predicates:

$$\exists ps,ps' : \mathsf{list}.ps.lsalaries * \mathsf{list}.ps'.lsalaries' \wedge \tfrac{\sum ps}{|ps|} = \tfrac{\sum ps'}{|ps'|}$$
$$\exists ps,ps' : \mathsf{list}.ps.lsalaries * \mathsf{list}.ps'.lsalaries' \wedge a_l = a_l'$$

Here, we use notation $|ps|$ for the length of list ps, and $\sum ps$ for the sum of all numbers in ps with $\sum[\,] = 0$.

The proof of $\{\mathbf{I}_{sl}(\mathcal{A})\}\ \mathsf{AV\_SAL}\ ;\ \mathsf{AV\_SAL}[\xi]\ \{\mathbf{I}_{sl}(=_{\{a_l\}})\}$ should not be too difficult to work out knowing that the loop invariant in $\mathsf{AV\_SAL}$ is

$$\begin{array}{l}\exists ps,ps' : \mathsf{list}.ps'.lsalaries' \\ \quad * \,(\ \mathsf{list}.ps.lsalaries \\ \qquad \wedge\ \exists ps_{mis},ps_{prev} : (ps = ps_{prev} +\!\!+\, ps_{mis}) \wedge \mathsf{list}.ps_{mis}.p \\ \qquad\qquad\qquad\qquad \wedge\ (s = \sum ps_{prev}) \wedge (n = |ps_{prev}|)\ \ ) \\ \quad \wedge\ (\frac{\sum ps}{|ps|} = \frac{\sum ps'}{|ps'|})\end{array}$$

A similar invariant corresponds to the loop in $\mathsf{AV\_SAL}[\xi]$ where program variables are now primed (i.e. renamed by ξ) and $\frac{\sum ps}{|ps|}$ must be changed by a_l. The following predicate correspond to the "intermediate" assertion, i.e. the precondition of $\mathsf{AV\_SAL}$ and postcondition of $\mathsf{AV\_SAL}[\xi]$:

$$\exists ps,ps' : \mathsf{list}.ps.lsalaries * \mathsf{list}.ps'.lsalaries' \wedge \big(a_l = \tfrac{\sum ps'}{|ps'|}\big)$$

Notice that the language for mutable data structure we use does not allow to manipulate pointer arithmetics nor to test on pointer values. In fact, the characterisation of security in separation logic as given above, only observes the values that pointers represent, not their actual address values. This is similar to Banerjee and Naumann's approach to non-interference in an object based setting [3]. If tests on pointers are allowed, this approach would let leak information throughout address values. An example of such leakage is program $\mathsf{if}\ y_h = 0\ \mathsf{then}\ p_l := q_l\ \mathsf{else}\ p_l := \mathsf{cons}(q_l.1, q_l.2)\ \mathsf{fi}$ with y_h being the only private variable. At the end of the program, p_l and q_l agree on the value they represent, although if $y_h = 0$ they point to the same address, otherwise they do not. We do not discard however a stronger characterisation of security that control (some) leakage through addresses.

8. Nondeterminism and CTL

Computation Tree Logic (CTL for short) [9] is a temporal logic that extends propositional logic with modalities to express properties on the branching structure of a nondeterministic execution. That is, CTL temporal operators allow to quantify over execution paths (i.e., maximal transition sequences leaving a particular state). Apart from the usual propositional operations (atomic propositions, $\neg$, $\vee$, $\wedge$, $\rightarrow$,...), CTL provides (unary) temporal operators EF, AF, EG, and AG. Formula $\mathsf{EF}\,\phi$ states that *exists* an execution path that leads to a *future* state in which ϕ holds, while $\mathsf{AF}\,\phi$ states that *all* execution paths lead to a *future* state in which ϕ holds. Dually, $\mathsf{EG}\,\phi$ states that *exists* an execution path in which ϕ *globally* holds (i.e., it holds in every state along this execution), and $\mathsf{AG}\,\phi$ says that for *all* paths ϕ holds *globally*. CTL includes other (more expressive) operators which we omit in this discussion.

Formally, a transition system $(\mathsf{Conf}, \rightsquigarrow)$ is extended with a function Prop that to each configuration in Conf assigns a set of atomic propositions. $\mathsf{Prop}(c)$ is the set of all atomic propositions valid in c. An *execution* is a maximal (finite or infinite) sequence of configurations $\rho = c_0 c_1 c_2 \ldots$ such that $c_i \rightsquigarrow c_{i+1}$ and if it ends in a configuration c_n then $c_n \not\rightsquigarrow$. For $i \geq 0$, let $\rho_i = c_i$ be the i-th state in ρ (if ρ is finite, $i+1$ must not exceed ρ's length).

Let $c \models \phi$ denote that CTL formula ϕ *holds* in configuration c. The semantics of CTL is defined by

$$c \models \mathsf{EF}\,\phi \quad \text{iff} \quad \exists \rho : \rho_0 = c : \exists i : \rho_i \models \phi$$
$$c \models \mathsf{AF}\,\phi \quad \text{iff} \quad \forall \rho : \rho_0 = c : \exists i : \rho_i \models \phi$$

AG and EG are the dual of EF and AF respectively, that is, $\mathsf{AG}\,\phi \equiv \neg\,\mathsf{EF}\,\neg\phi$ and $\mathsf{EG}\,\phi \equiv \neg\,\mathsf{AF}\,\neg\phi$. For an atomic proposition p, $c \models p$ iff $p \in \mathsf{Prop}(c)$. The semantics of the propositional operators $\neg$, $\wedge$, $\vee$, $\rightarrow$ are as usual (e.g., $c \models \phi \wedge \psi$ iff $c \models \phi$ and $c \models \psi$).

Let end be the atomic proposition that indicates that the execution reaches a successfully terminating state, i.e., $\mathsf{end} \in \mathsf{Prop}(S, \mu)$ iff $S = \surd$. Let mid indicate that program $S[\xi]$ is about to be executed, i.e., $\mathsf{mid} \in \mathsf{Prop}(S', \mu)$ iff $S' = S[\xi]$. Let $\mathsf{Ind}[\mathcal{I}]$ be an atomic proposition indicating indistinguishability in a state. Thus $\mathsf{Ind}[\mathcal{I}] \in \mathsf{Prop}(S, \mu)$ iff $\mu \sim_{\xi}^{\mathcal{I}} \mu$. We let $S \models \Phi$ denotes $\forall \mu : (S, \mu) \models \Phi$. For the sake of simplicity, we consider simple memories as in Example 1. (More complex states are possible, but it will be necessary to introduce additional atomic propositions to characterise separable memories like we did in the Section 7.)

In the following we give characterisations of noninterference in CTL. A program S is TS $(\mathcal{I}_1, \mathcal{I}_2)$-secure if and only if $S \triangleright S[\xi]$ satisfies $\mathsf{Ind}[\mathcal{I}_1] \rightarrow \mathsf{AG}(\mathsf{mid} \rightarrow \mathsf{EF}(\mathsf{end} \wedge \mathsf{Ind}[\mathcal{I}_2]))$, which says that "whenever the initial state is indistinguishable, every time $S[\xi]$ is reached (and hence S terminates) then there is an execution that leads to a terminating indistinguishable state".

For the termination insensitive case, first notice that a program does not terminate if no execution reaches a terminating state. That is, $\neg\exists\mu' : (S, \mu) \rightsquigarrow^* (\surd, \mu')$, or equivalently $\forall S', \mu' : (S, \mu) \rightsquigarrow^* (S', \mu') : S' \neq \surd$, and hence $(S, \mu) \models \mathsf{AG}\,\neg\mathsf{end}$. Now, program S is TI $(\mathcal{I}_1, \mathcal{I}_2)$-secure if and only if $S \triangleright S[\xi]$ satisfies $\mathsf{Ind}[\mathcal{I}_1] \rightarrow \mathsf{AG}(\mathsf{mid} \rightarrow ((\mathsf{AG}\,\neg\mathsf{end}) \vee \mathsf{EF}(\mathsf{end} \wedge \mathsf{Ind}[\mathcal{I}_2])))$, that is, "if the initial state is indistinguishable then, every time $S[\xi]$ is reached, the program does not terminate or there is an execution that leads to a terminating indistinguishable state". The CTL characterisations of security can be proven correct using Corollary 1 (see Appendix D).

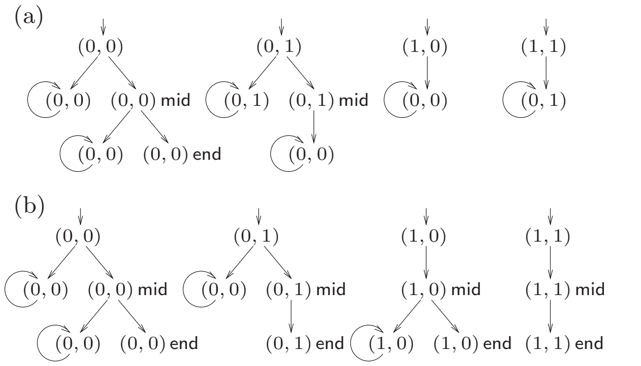

Figure 2. Automata for programs of Example 6

Example 6. Let y_h be a confidential variable in the following programs (borrowed from [22]):

(a)
```
if [] y_h=0 →
      y_h := y_h
   [] true →
      while true do y_h := 0 od
fi
```

(b)
```
if [] y_h=0 →
      while true do y_h := 0 od
   [] true →
      y_h := y_h
fi
```

We check whether they are (possibilistic) noninterfering [38, 22], that is, whether they are $(=_L, =_L)$-secure. We use CTL and for this we set $\mathsf{Ind}[=_\emptyset] \equiv \mathsf{true}$. The automata of the (self-composed) programs (a) and (b) are depicted in Figure 2. In the picture, variables take only value 0 or 1. Besides, a state is depicted with a tuple (d, d') containing the values of y_h and y'_h respectively. Labels mid and end next to a state indicate that they hold in this state. Initial states are indicated with a small incoming arrow.

Observe that both programs satisfy the TI formula $\mathsf{true} \rightarrow \mathsf{AG}(\mathsf{mid} \rightarrow ((\mathsf{AG}\,\neg\mathsf{end}) \vee \mathsf{EF}(\mathsf{end} \wedge \mathsf{true})))$. As observed in [22], program (a) *does* leak information: if it terminates, it does it with $y_h = 0$. The TS formula $\mathsf{true} \rightarrow \mathsf{AG}(\mathsf{mid} \rightarrow \mathsf{EF}(\mathsf{true} \wedge \mathsf{end}))$ detects such leakage. Notice that the second automaton from the left has an execution that completes its "first phase" but never terminates. The formula is valid in program (b).

LTL and possibilistic security. A similar characterisation can be given for LTL [24] but limited to deterministic programs. Like CTL, LTL extends propositional logic with modal operations. However this modalities refer only to properties of single executions disregarding quantification. LTL provides (unary) temporal operators F and G. $\mathsf{F}\,\phi$ holds in a program execution if ϕ holds in the *future*, i.e. in some suffix of this execution. $\mathsf{G}\,\phi$ holds in a program execution if ϕ holds *globally*, i.e. in all suffixes of this execution.

In a deterministic setting, the semantics of F and G can be characterised in terms of reachability: $c \models \mathsf{F}\,\phi$ iff $(\exists c' : c \rightsquigarrow^* c' : c' \models \phi)$, and $c \models \mathsf{G}\,\phi$ iff $(\forall c' : c \rightsquigarrow^* c' : c' \models \phi)$. Using Corollary 2, TS and TI $(\mathcal{I}_1, \mathcal{I}_2)$-security can be characterised in LTL respectively by formulas $\mathsf{Ind}[\mathcal{I}_1] \rightarrow ((\mathsf{F}\,\mathsf{mid}) \rightarrow \mathsf{F}(\mathsf{end} \wedge \mathsf{Ind}[\mathcal{I}_2]))$, and $\mathsf{Ind}[\mathcal{I}_1] \rightarrow \mathsf{G}(\mathsf{end} \rightarrow \mathsf{Ind}[\mathcal{I}_2])$.

It is known that CTL and LTL are incomparable on expressiveness. $\mathsf{AG}(\phi \rightarrow \mathsf{EF}\,\psi)$ is a typical CTL formula which is not expressible in LTL. It can be shown that $\mathsf{AG}(\phi \rightarrow (\mathsf{AG}\,\psi \vee \mathsf{EF}\,\psi))$ is neither. These formulas occur as nontrivial subformulas of the CTL characterisations of security. As a consequence, security cannot be characterised with LTL in a non-deterministic setting (at least using our technique).

More generally, our encoding of possibilistic security is limited to logics that can observe branching structure of executions such as CTL, CTL*, or the modal μ-calculus, ruling out other logics usually used on concurrent systems which can only observe linear behaviour (this includes not only LTL but also Owicki-Gries logic).

Termination and possibilistic security. Example 6 anticipates certain subtleties arising from termination. It has been argued that program (b) still leaks information [22]. A sharp adversary that can observe *possibilistic* non-termination may detect that a possible execution of the same instance of a program (i.e. running with the same starting memory) stalls indefinitely. Such adversary can observe a difference between program (b) under $y_h = 0$ (which sometimes terminates and some others does not) or under $y_h = 1$ (which always terminates). To this extend, our characterisation of TS $(\mathcal{I}_1, \mathcal{I}_2)$-security fails.

So far, we have considered *strict* non-termination: $(S, \mu)\bot$ states that S *does not* terminate in μ. A notion of *possibilistic* termination can be given, denoted by $(S, \mu)\nearrow$, that states that there is an execution of S from memory μ that does not terminate. I.e., $(S, \mu)\nearrow$ iff there exists ρ such that $(S, \mu){=}\rho_0$ and $\forall i : i \geq 0 : \neg\exists\mu' : \rho_i{=}(\surd, \mu')$.

From Definition 1, S is (TS) $(\mathcal{I}_1, \mathcal{I}_2)$-secure if for all μ_1, μ_2 such that $\mu_1 \sim^{\mathcal{I}_1}_{id_{V_1}} \mu_2$,

$$(o) \quad \forall\mu'_1 : (S, \mu_1) \rightsquigarrow^* (\surd, \mu'_1) \\ \Rightarrow (\exists\mu'_2 : (S, \mu_2) \rightsquigarrow^* (\surd, \mu'_2) \wedge \mu'_1 \sim^{\mathcal{I}_2}_{id_{V_2}} \mu'_2.$$

In addition to this, one of the following termination conditions can be also required:

$$(i)\ (S, \mu_1)\bot \Rightarrow (S, \mu_2)\bot \qquad (ii)\ (S, \mu_1)\nearrow \Rightarrow (S, \mu_2)\bot$$
$$(iii)\ (S, \mu_1)\bot \Rightarrow (S, \mu_2)\nearrow \qquad (iv)\ (S, \mu_1)\nearrow \Rightarrow (S, \mu_2)\nearrow$$

Since $\neg(S, \mu)\bot$ iff $\exists\mu' : (S, \mu) \rightsquigarrow^* (\surd, \mu')$, and provided that $\mathcal{I}_1$ is symmetric, (i) can be deduced from (o). Since (o) implies (i), and $(S, \mu)\bot$ implies $(S, \mu)\nearrow$, then (ii) is redundant as well.

Condition (iii) states that if a program *may* not terminate then it *must* not terminate in any indistinguishable state. As a consequence it considers insecure any program that sometimes terminates and some other does not. In particular, program (b) in Example 6 is insecure under this condition. But so is

$$\mathsf{if}\ []\ \mathsf{true} \rightarrow \mathsf{while}\ \mathsf{true}\ \mathsf{do}\ h := h\ \mathsf{od}\ []\ \mathsf{true} \rightarrow h := h\ \mathsf{fi} \qquad (2)$$

which evidently does not reveal any information.

Condition (iv) states that a program that may not terminate in a given state, should be able to reach a non-termination situation in any indistinguishable state. Provided that $\mathcal{I}_1$ is symmetric, this also means that a secure program that surely terminates in a state, surely terminates in any indistinguishable state. This definition rules out Example 6(b) as insecure, but considers (2) to be secure.

The following CTL formulas characterise these restrictions:

$$\begin{array}{ll} (iii) & \mathsf{Ind}[\mathcal{I}_1] \rightarrow \big((\mathsf{EG}\,\neg\mathsf{mid}) \rightarrow \mathsf{AG}\,\neg\mathsf{end} \big) \\ (iv) & \mathsf{Ind}[\mathcal{I}_1] \rightarrow \big((\mathsf{EG}\,\neg\mathsf{mid}) \rightarrow \mathsf{AG}(\mathsf{mid} \rightarrow \mathsf{EG}\,\neg\mathsf{end}) \big) \\ (iv^s) & \mathsf{Ind}[\mathcal{I}_1] \rightarrow \big((\mathsf{AF}\,\mathsf{mid}) \rightarrow \mathsf{AF}\,\mathsf{end} \big) \end{array}$$

where (iv^s) is the restriction of (iv) to the case in which $\mathcal{I}_1$ is symmetric. Notice that (iii) is not satisfied in any automaton of Figure 2(b), (iv) is not satisfied by the second automaton from the left, and (iv^s), by the third.

9. Discussion

9.1. Benefits of completeness. While our characterization of non-interference with Hoare logics is sound and complete, it is undecidable in general and not obviously compositional. This is to be contrasted with type systems for non-interference, as type inference is usually decidable and the typing rules is compositional.

Completeness of the Hoare logic allows us to achieve the best of the two worlds. Indeed, consider the simple imperative language of the introduction and let P

be a program with low variables $\vec{x}$ and with high variables $\vec{y}$, and let $[\vec{x}', \vec{y}'/\vec{x}, \vec{y}]$ be a renaming of the program variables of P with fresh variables; it follows immediately from the soundness of the type system of [42] and from our characterization of non-interference that the following rule is valid:

$$\frac{\vec{y} : \mathsf{high},\ \vec{x} : \mathsf{low} \vdash P : \tau\ \mathsf{cmd}}{\{\vec{x} = \vec{x}'\}\ P; (P[\vec{x}', \vec{y}'/\vec{x}, \vec{y}])\ \{\vec{x} = \vec{x}'\}}$$

Likewise, one can show the validity of rules such as:

$$\frac{\begin{array}{c}\{\vec{x} = \vec{x}'\}\ P; P[\vec{x}', \vec{y}'/\vec{x}, \vec{y}]\ \{\vec{x} = \vec{x}'\} \\ \{\vec{x} = \vec{x}'\}\ Q; Q[\vec{x}', \vec{y}'/\vec{x}, \vec{y}]\ \{\vec{x} = \vec{x}'\}\end{array}}{\{\vec{x} = \vec{x}'\}\ (P; Q); (P; Q)[\vec{x}', \vec{y}'/\vec{x}, \vec{y}]\ \{\vec{x} = \vec{x}'\}}$$

Such rules are most useful in order to automate or shorten proofs of non-interference for programs.

9.2. Model Checking. Results of Section 8 allow to directly use model checker tools such as SPIN [40] or SMV [39] for checking $(\mathcal{I}_1, \mathcal{I}_2)$-security. However, this is limited to programs with finite state spaces.

To model check infinite state space systems, we will need either to use the assistance of proof checkers, or to resort to abstractions. $(\mathcal{I}_1, \mathcal{I}_2)$-security already suggests some abstraction in predicates $\mathcal{I}_1$ and $\mathcal{I}_2$. Depending on the granularity defined by these predicates, abstractions may be complete, or further abstraction may be necessary resulting, in any case, in safe approximations. Since $(\mathcal{I}_1, \mathcal{I}_2)$-security is a particular kind of CTL or LTL property, we hope that model checking can be specifically tailored to it.

9.3. Other forms of non-interference. One natural direction for further research is to provide similar characterizations for other notions of non-interference. In particular, we intend to study further the appropriateness of our approach for capturing declassification.

Further, we intend to extend our results to account for covert channels. In particular, we expect that our characterizations may be adapted to probabilistic non-interference, a refined notion of non-interference that is adopted in many works on non-deterministic languages, see e.g. [37, 38], and that eliminates probabilistic covert channels by considering the distribution of the inputs and outputs. It will also be interesting to assess the applicability of probabilistic Hoare logics [12] and probabilistic model-checking [4] in this context. Likewise, we hope that our characterizations may be adapted to account for timing leaks, using appropriate logics for timed systems [18].

9.4. Extension to Java. Another natural direction for further research is to extend the results of this paper to other language constructs such as exceptions and objects. We are particularly interested in applying our method to Java applets, in particular since security policies for JavaCard applets often require some form of non-interference [26], and type-based systems such as the ones of [3] seem too restrictive for accepting JavaCard applets as secure (of course, one could resort to JFlow [27] but we are not aware of any non-interference result for this type system).

The extension of our work to Java raises subtle issues, in particular with respect to definability of heap indistinguishability [3] that involves a partial bijection between addresses, and with exceptions that poses problems similar to termination-sensitive non-interference. However, we hope that it is possible to give an appropriate characterization of non-interference in the Java Modeling Language JML [23], and use existing verification tools such as Jack [7] or Bogor [32] (see [6] for an overview of JML tools) to validate non-interference.

9.5. Related work. A large body of recent works on non-interference follows a type-based approach, see [35] for a recent survey. There are however, notable exceptions to this trend. In particular, many characterizations of non-interference, often amenable to model-checking techniques, have been developed in the context of process algebras, see e.g. [2, 13, 34]. In fact [14] reports on a model checker for SPA [13], though the approach is based on bisimulation checking rather than verification of temporal properties.

Closest to our concerns is the work of Joshi and Leino [22], who provide a characterization of non-interference using weakest precondition calculi. Like ours, their characterization can be applied to a variety of programming constructs, including non-deterministic constructs, and can handle termination sensitive non-interference. Their use of cylinders eliminates the need to resort to self-composition; on the other hand, their approach is circumscribed to weakest precondition calculi.

Pursuing the line of work initiated by Joshi and Leino, Darvas, Hähnle and Sands [11] rely on dynamic logic to provide criteria for non-interference (TS and TI, and modulo declassification) in the context of JavaCard. While they suggest to use self-composition as a means to guarantee non-interference, they do not show that their criterion is sound and complete, nor do they highlight the benefits of completeness. More recently, Jacobs and Warnier [21] provide a method to verify non-interference for (sequential) Java programs. Their method relies on a relational Hoare logic for JML programs, and is applied to an example involving logging in a cash register. However there lacks a precise analysis of the form of non-interference enforced by their method. None of [11, 21] handle non-determinism.

Generalisations of non-interference were given elsewhere [33, 25, 15, 36]. In particular, [15] has recently provided a definition of secrecy much like ours $(\mathcal{I}, \mathcal{I}')$-security. However, they focus on abstract interpretation for the anaysis of the property.

Acknowledgments: We thank the anonymous reviewers for providing valuable comments.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In POPL'99 [29], pages 147–160.

[2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[3] A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Submitted for publication*, 2003.

[4] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *Proc.* 15^{th} *FSTTCS,* Pune, India, volume 1026 of *LNCS*, pages 499–513. Springer-Verlag, 1995.

[5] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Comput. Sci.*, 281(1):109–130, 2002.

[6] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Proc. of FMICS'03*, volume 80 of *ENTCS*. Elsevier Publishing, 2003.

[7] L. Burdy, A. Requet, and J.-L. Lanet. Java Applet Correctness: a Developer-Oriented Approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proceedings of FME'03*, number 2805 in LNCS, pages 422–439. Springer-Verlag, 2003.

[8] D. Clark, C. Hankin, and S. Hunt. Information flow for Algol-like languages. *Journal of Computer Languages*, 2002.

[9] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Prog. Lang. Syst.*, 8(2):244–263, 1986.

[10] S. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978.

[11] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Workshop on Issues in the Theory of Security, WITS'03,* Warsaw, 2003.

[12] J. den Hartog and E. de Vink. Verifying probabilistic programs using a hoare-like logic. *International Journal of Foundations of Computer Science*, 13, 2002. 315–340.

[13] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *J. of Computer Security*, 3(1):5–33, 1995.

[14] R. Focardi and R. Gorrieri. Automatic compositional verification of some security properties. In T. Margaria and B. Steffen, editors, *Proc. of TACAS'96,* Passau, volume 1055 of *LNCS*, pages 111–130. Springer-Verlag, Apr. 1996.

[15] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. of the* 31^{th} *ACM Symp. on Principles of Programming Languages,* Venice, pages 186–197, 2004.

[16] J. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security abd Privacy*, pages 11–20. IEEE Comp. Soc. Press, 1982.

[17] N. Heinze and J. Riecke. The SLam calculus: programming with secrecy and integrity. In POPL'98 [28], pages 365–377.

[18] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.

[19] C. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580,583, 1969.

[20] S. Ishtiaq and P. O'Hearn. Bi as an assertion language for mutable data structures. In *Proc. of the* 28^{th} *ACM Symp. on Principles of Programming Languages,* London, pages 14–26, 2001.

[21] B. Jacobs and M. Warnier. Formal proofs of confidentiality in java programs, 2003. Manuscript.

[22] R. Joshi and K. Leino. A semantic approach to secure information flow. *Sci. Comput. Programming*, 37:113–138, 2000.

[23] G. Leavens, A. Baker, and C. Ruby. Preliminary Design of JML: a Behavioral Interface Specification Language for Java. Technical Report 98-06, Iowa State University, Department of Computer Science, 1998.

[24] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.

[25] H. Mantel. Unwinding Possibilistic Security Properties. In F. Cuppens et al, editor, *Proc. of ESORICS 2000,* Toulouse, volume 1895 of *LNCS*, pages 238–254. Springer-Verlag, 2000.

[26] R. Marlet and D. L. Métayer. Security properties and java card specificities to be studied in the secsafe project, 2001. Deliverable SECSAFE-TL-006.

[27] A. C. Myers. JFlow: Practical mostly-static information flow control. In POPL'99 [29], pages 228–241.

[28] *Proc. of the* 25^{th} *ACM Symp. on Principles of Programming Languages,* San Diego, 1998.

[29] *Proc. of the* 26^{th} *ACM Symp. on Principles of Programming Languages,* San Antonio, 1999.

[30] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. of the* 29^{th} *ACM Symp. on Principles of Programming Languages,* Portland, pages 319–330, 2002.

[31] J. Reynolds. Intuitionistic reasoning about shared mutable data structure. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, 2000.

[32] Robby, E. Rodríguez, M. Dwyer, and J. Hatcliff. Checking strong specifications using an extensible software model checking framework. In K. Jensen and A. Podelski, editors, *Proc. of TACAS 2004*, Barcelona, volume 2988 of *LNCS*, pages 404–420. Springer-Verlag, Apr. 2004.

[33] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI, 1992.

[34] P. Ryan and S. Schneider. Process algebra and non-interference. *J. of Computer Security*, 9, 2001.

[35] A. Sabelfeld and A. Myers. Language-based information flow security. *IEEE J. on Selected Areas in Communications*, 21(1):5–19, 2003.

[36] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, LNCS. Springer-Verlag, 2004. To appear.

[37] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher Order and Symbolic Computation*, 14(1):59–91, Mar. 2001.

[38] G. Smith and D. Volpano. Secure information flow in multi-threaded imperative languages. In POPL'98 [28], pages 355–364.

[39] SMV: Symbolic model verifier. Cadence SMV: `www-cad.eecs.berkeley.edu/~kenmcmil/smv/`, CMU SMV: `www.cs.cmu.edu/~modelcheck/smv.html`, NuSMV: `nusmv.irst.itc.it/`.

[40] SPIN. `spinroot.com/spin/whatispin.html`.

[41] D. Volpano and G. Smith. A type based approach to program security. In M. Bidoit and M. Dauchet, editors, *Proc. of TAPSOFT'97*, volume 1214 of *LNCS*, pages 607–621. Springer-Verlag, 1997.

[42] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. of Computer Security*, 4(3):167–187, 1996.

[43] S. Zdancewic and A. Myers. Observational determinism for concurrent program security. In *Proc. of the* 16^{th} *IEEE Computer Security Foundations Workshop* Pacific Grove, pages 29–43. IEEE Comp. Soc. Press, 2003.

A. Formalization of the Assumptions

Assumption 1. *Transitions preserve the set of variables of a program. Moreover they do not affect the values of other variables than those appearing in the program*: For all S, S', μ_1, μ_2, and μ', if $\mathsf{var}(S) = \mathsf{var}(\mu_1)$ and $(S, \mu_1 \oplus \mu_2) \rightsquigarrow (S', \mu')$, then $\mathsf{var}(S) \supseteq \mathsf{var}(S')$ and $\exists \mu'_1 : \mu' = \mu'_1 \oplus \mu_2$. □

A relation $R \subseteq (\mathsf{Conf} \times \mathsf{Conf})$ is a *bisimulation* if it is symmetric and for all $\langle (S_1, \mu_1), (S_2, \mu_2) \rangle \in R$ the following properties hold: (a) $S_1 = S_2$; (b) $\forall x \in \mathsf{var}(S_1) : \mathsf{v}(\mu_1, x) = \mathsf{v}(\mu_2, x)$; and (c) $\forall c_1 : (S_1, \mu_1) \rightsquigarrow c_1 \Rightarrow \exists c_2 : (S_2, \mu_2) \rightsquigarrow c_2 \wedge \langle c_1, c_2 \rangle \in R$.

Assumption 2. *Apart from its syntax, the semantics of a program depends only on the value of its own variables. Moreover, given a memory, it is always possible to find another one with the same values that can be separated in two parts, one of which contains exactly all information relevant to the program*: Relation

$$\{\langle (S, \mu_1), (S, \mu_2) \rangle \mid \forall x \in \mathsf{var}(S) : \mathsf{v}(\mu_1, x) = \mathsf{v}(\mu_2, x)\}$$

is a bisimulation. Moreover for every memory μ there are μ_1, μ_2 such that $\mathsf{var}(\mu_1) = \mathsf{var}(S)$ and $\forall x : \mathsf{v}(\mu, x) = \mathsf{v}(\mu_1 \oplus \mu_2, x)$ (and hence, $\langle (S, \mu), (S, \mu_1 \oplus \mu_2) \rangle$ belongs to the relation). □

As a consequence of Assumption 2, relation

$$\{\langle (S, \mu \oplus \mu_1), (S, \mu \oplus \mu_2) \rangle \mid (S \neq \surd \Rightarrow \mathsf{var}(\mu) \supseteq \mathsf{var}(S)) \wedge \mu \oplus \mu_1 \text{ and } \mu \oplus \mu_2 \text{ are defined} \} \quad (3)$$

is also a bisimulation. This property means that the behaviour of S is the same regardless the contents of the piece of memory not touched by variables in $\mathsf{var}(S)$. (Notice that bisimulation (3) and the one of Assumption 2 coincides under memories of Example 1.)

Assumption 3. *The operational semantics of the language* Lang *is independent of variable names*: Relation

$$\{\langle (S, \mu), (S[y/x], \mu[x \mapsto d][y \mapsto \mathsf{v}(\mu, x)]) \rangle \mid y \notin \mathsf{var}(S) \wedge S \in \mathsf{Lang} \cup \{\surd\} \wedge d \text{ is a value of } x\text{'s type}\}$$

is a bisimulation. □

B. Proof of Theorem 1

(a) Termination sensitive case.

Case ($\Rightarrow$). Let $\mathsf{var}(\mu_1) = \mathsf{var}(\mu'_1) = \mathsf{var}(S_1)$ and $\mathsf{var}(\mu_2) = \mathsf{var}(S_2)$, and suppose $\mu_1 \oplus \mu_2 \sim^{\mathcal{I}}_{\phi} \mu_1 \oplus \mu_2$ and $(S_1 \triangleright S_2, \mu_1 \oplus \mu_2) \rightsquigarrow^* (S_2, \mu'_1 \oplus \mu_2)$. By property (a) of $\triangleright$, $(S_1, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu'_1 \oplus \mu_2)$. Since $S_1 \approx^{\phi; \mathcal{I}}_{\phi'; \mathcal{I}'} S_2$, there exists μ' such that

$$(S_2, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu') \quad \text{and} \quad \mu'_1 \oplus \mu_2 \sim^{\mathcal{I}'}_{\phi'} \mu'.$$

But, by Fact 1.1, $\mu' = \mu_1 \oplus \mu'_2$ for some μ'_2, and hence $\mu'_1 \oplus \mu_2 \sim^{\mathcal{I}'}_{\phi'} \mu_1 \oplus \mu'_2$. By Fact 1.2

$$(S_2, \mu'_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu'_1 \oplus \mu'_2) \quad \text{and} \quad \mu'_1 \oplus \mu_2 \sim^{\mathcal{I}'}_{\phi'} \mu_1 \oplus \mu'_2.$$

Finally, since $\phi : \mathsf{var}(S_1) \to \mathsf{var}(S_2)$ and $\mathsf{var}(S_i) = \mathsf{var}(\mu_i) = \mathsf{var}(\mu'_i)$, $i \in \{1, 2\}$, $\mu'_1 \oplus \mu_2 \sim^{\mathcal{I}'}_{\phi'} \mu_1 \oplus \mu'_2$ implies

$$\mu'_1 \oplus \mu'_2 \sim^{\mathcal{I}'}_{\phi'} \mu'_1 \oplus \mu'_2$$

by Proposition 1, concluding this part of the proof.

Case ($\Leftarrow$). By Fact 1.6, there is always possible to find equivalent separable memory. So, w.l.o.g., take $\mu_1{\oplus}\mu_2 \sim^{\mathcal{I}}_{\phi} \eta_1{\oplus}\eta_2$ $(S_1, \mu_1{\oplus}\mu_2) \rightsquigarrow^* (\surd, \mu')$. By Fact 1.1, there is μ_1' such that $\mu' = \mu_1'{\oplus}\mu_2$. Therefore,

$$\mu_1{\oplus}\mu_2 \sim^{\mathcal{I}}_{\phi} \eta_1{\oplus}\eta_2 \quad \text{and} \quad (S_1, \mu_1{\oplus}\mu_2) \rightsquigarrow^* (\surd, \mu_1'{\oplus}\mu_2)$$

By Proposition 1 and Fact 1.2,

$$\mu_1{\oplus}\eta_2 \sim^{\mathcal{I}}_{\phi} \mu_1{\oplus}\eta_2 \quad \text{and} \quad (S_1, \mu_1{\oplus}\eta_2) \rightsquigarrow^* (\surd, \mu_1'{\oplus}\eta_2)$$

By property (a) of $\triangleright$, $(S_1 \triangleright S_2, \mu_1{\oplus}\eta_2) \rightsquigarrow^* (S_2, \mu_1'{\oplus}\eta_2)$. Since $S_1 \stackrel{\triangleright\,\phi,\mathcal{I}}{\approx}_{\phi',\mathcal{I}'} S_2$ is assumed to hold, this implies

$$(S_2, \mu_1'{\oplus}\eta_2) \rightsquigarrow^* (\surd, \mu_1'{\oplus}\eta_2') \text{ and } \mu_1'{\oplus}\eta_2' \sim^{\mathcal{I}'}_{\phi'} \mu_1'{\oplus}\eta_2'.$$

for some η_2'. By Proposition 1 and Fact 1.2

$$(S_2, \eta_1{\oplus}\eta_2) \rightsquigarrow^* (\surd, \eta_1{\oplus}\eta_2') \text{ and } \mu_1'{\oplus}\mu_2 \sim^{\mathcal{I}'}_{\phi'} \eta_1{\oplus}\eta_2'.$$

which concludes the termination sensitive case.

(b) Termination insensitive case.

Case ($\Rightarrow$). Let $\mathsf{var}(\mu_1) = \mathsf{var}(\mu_1') = \mathsf{var}(S_1)$ and $\mathsf{var}(\mu_2) = \mathsf{var}(S_2)$, and suppose $\mu_1{\oplus}\mu_2 \sim^{\mathcal{I}}_{\phi} \mu_1{\oplus}\mu_2$ and $(S_1 \triangleright S_2, \mu_1{\oplus}\mu_2) \rightsquigarrow^* (S_2, \mu_1'{\oplus}\mu_2)$. By Property (a) of $\triangleright$, $(S_1, \mu_1{\oplus}\mu_2) \rightsquigarrow^* (\surd, \mu_1'{\oplus}\mu_2)$. Since we assume 1 holds, either (i) $(S_2, \mu_1{\oplus}\mu_2) \rightsquigarrow^* (\surd, \mu')$ and $\mu_1'{\oplus}\mu_2 \sim^{\mathcal{I}}_{\phi} \mu'$, for some μ', or (ii) $(S_2, \mu_1{\oplus}\mu_2)\bot$. Case (i) follows like for case (a). So suppose case (ii) holds. But then $(S_2, \mu_1'{\oplus}\mu_2)\bot$ by Fact 1.3.

Case ($\Leftarrow$). Like for case (a), using Fact 1.1 and 6, suppose w.l.o.g. that $\mu_1{\oplus}\eta_2 \sim^{\mathcal{I}}_{\phi} \mu_1{\oplus}\eta_2$ and $(S_1 \triangleright S_2, \mu_1{\oplus}\eta_2) \rightsquigarrow^* (S_2, \mu_1'{\oplus}\eta_2)$ with μ_1, μ_2, η_1, and η_2 as before. Since $S_1 \stackrel{\triangleright\,\phi,\mathcal{I}}{\approx}_{\phi',\mathcal{I}'} S_2$ is assumed to hold, either (i) $(S_2, \mu_1'{\oplus}\eta_2) \rightsquigarrow^* (\surd, \mu_1'{\oplus}\eta_2')$ and $\mu_1'{\oplus}\eta_2' \sim^{\mathcal{I}}_{\phi} \mu_1'{\oplus}\eta_2'$, for some η_2' or (ii) $(S_2, \mu_1'{\oplus}\eta_2)\bot$. Case (i) follows as in case ($\Leftarrow$) of (a). So suppose case (ii) holds. But then $(S_2, \eta_1{\oplus}\eta_2)\bot$ by Fact 1.3, which concludes the proof.

C. Characterisation of security in Separation Logic: Proofs

We prove first that $\mathbf{I}_{sl}(\mathcal{I})$ characterises indistinguishability in a separable memory, i.e., a memory μ such that $\exists \mu_1, \mu_2 : \mu = \mu_1{\oplus}\mu_2$ with $\mathsf{var}(\mu_1) = \mathsf{var}(S)$ and $\mathsf{var}(\mu_2) = \mathsf{var}(S)'$. First, notice that $\overline{\mathsf{v}}(h, l)$ is defined and $l \neq \mathsf{nil}$ iff $\mathrm{reach}(\{l\}, h) \subseteq \mathrm{dom}(h)$. Let $\overline{\mathsf{v}}(h_1, s(\vec{x})) = (\overline{\mathsf{v}}(h_1, s(x_1)), .., \overline{\mathsf{v}}(h_1, s(x_n)))$ and similarly for $\overline{\mathsf{v}}(h_2, s(\vec{x}'))$. As a consequence, if $\mathrm{dom}(s) = \mathsf{var}(S) \cup \mathsf{var}(S)'$,

$\exists h_1, h_2 : h = h_1{\oplus}h_2 : \overline{\mathsf{v}}(h_1, s(\vec{x}))$ and $\overline{\mathsf{v}}(h_2, s(\vec{x}'))$ are defined
iff $\{s$ is a function with $\mathrm{dom}(s) = \mathsf{var}(S) \cup \mathsf{var}(S)'\}$
$\exists s_1, s_2, h_1, h_2 : h = h_1{\oplus}h_2 \wedge s = s_1{\oplus}s_2$
$\wedge\, \mathrm{dom}(s_1) = \mathsf{var}(S) \wedge \mathrm{dom}(s_2) = \mathsf{var}(S)'$
$\wedge\, \overline{\mathsf{v}}(h_1, s(\vec{x}))$ and $\overline{\mathsf{v}}(h_2, s(\vec{x}'))$ are defined
iff {Observation above}
$\exists s_1, s_2, h_1, h_2 : h = h_1{\oplus}h_2 \wedge s = s_1{\oplus}s_2$
$\wedge\, \mathrm{dom}(s_1) = \mathsf{var}(S) \wedge \mathrm{dom}(s_2) = \mathsf{var}(S)'$
$\wedge\ \mathrm{reach}(\mathrm{ran}(s_i) \cap \mathsf{Loc} - \{\mathsf{nil}\}, h_i) \subseteq \mathrm{dom}(h_i)$ for $i \in \{1, 2\}$
iff {Def. of $\oplus$ and $\mathsf{var}(s_i, h_i) = \mathrm{dom}(s_i)$}
$\exists s_1, s_2, h_1, h_2 : (s, h) = (s_1, h_1){\oplus}(s_2, h_2)$
$\quad \wedge\, \mathsf{var}(s_1, h_1) = \mathsf{var}(S) \wedge \mathsf{var}(s_2, h_2) = \mathsf{var}(S)'$ (4)

We now prove the correctness of $\mathbf{I}_{sl}(\mathcal{I})$:

$(s, h) \models \mathbf{I}_{sl}(\mathcal{I})$
iff {Semantics}
$\exists \vec{xs}, \vec{xs}' :$
$\exists h_1, h_2 : h = h_1{\oplus}h_2 : (\forall i : 1 \leq i \leq n : xs_i = \overline{\mathsf{v}}(h_1, s(x_i)))$
$\qquad \wedge (\forall i : 1 \leq i \leq n : xs_i' = \overline{\mathsf{v}}(h_2, s(x_i')))$
$\wedge\, (s, h) \models \mathbf{I}_v(\vec{xs}, \vec{xs}', \mathcal{I})$
iff {Def. of $\mathbf{I}_v$ and equality on vectors}
$\exists \vec{xs}, \vec{xs}' :$
$\exists h_1, h_2 : h = h_1{\oplus}h_2 : \vec{xs} = \overline{\mathsf{v}}(h_1, s(\vec{x})) \wedge \vec{xs}' = \overline{\mathsf{v}}(h_1, s(\vec{x}'))$
$\wedge\, (\langle \mathsf{v}(\mu, \vec{y}), \vec{xs}\rangle, \langle \mathsf{v}(\mu, \vec{y}'), \vec{xs}'\rangle) \in \mathcal{I}$
iff $\{\overline{\mathsf{v}}(h_1, s(x)){=}\mathsf{v}((s, h), x)$ and $\overline{\mathsf{v}}(h_2, s(x')){=}\mathsf{v}((s, h), x')\}$
$\exists \vec{xs}, \vec{xs}' :$
$\exists h_1, h_2 : h = h_1{\oplus}h_2 : \vec{xs} = \overline{\mathsf{v}}(h_1, s(\vec{x})) \wedge \vec{xs}' = \overline{\mathsf{v}}(h_1, s(\vec{x}'))$
$\wedge\, (\langle \mathsf{v}(\mu, \vec{y}), \mathsf{v}(\mu, \vec{x})\rangle, \langle \mathsf{v}(\mu, \vec{y}'), \mathsf{v}(\mu, \vec{x}')\rangle) \in \mathcal{I}$
iff $\{\exists v : f(z) = v$ iff $f(z)$ is defined, and Def. of $\sim^{\mathcal{I}}_{\xi}\}$
$\exists h_1, h_2 : h = h_1{\oplus}h_2 : \overline{\mathsf{v}}(h_1, s(\vec{x}))$ and $\overline{\mathsf{v}}(h_2, s(\vec{x}'))$ are defined
$\wedge\, (s, h) \sim^{\mathcal{I}}_{\xi} (s, h)$
iff { Remark (4) }
$(s, h) \sim^{\mathcal{I}}_{\xi} (s, h) \wedge$
$\exists s_1, s_2, h_1, h_2 : (s, h) = (s_1, h_1){\oplus}(s_2, h_2)$
$\quad \wedge\, \mathsf{var}(s_1, h_1) = \mathsf{var}(S) \wedge \mathsf{var}(s_2, h_2) = \mathsf{var}(S)'$

In the following we prove correctness and completeness of the characterisation in Separation Logic.

S is TI $(\mathcal{I}_1, \mathcal{I}_2)$-secure
iff {Theorem 3.2}
$\forall \mu_1, \mu_2, \mu_1', \mu_2' :$
$\mathsf{var}(\mu_1) = \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge\ \mathsf{var}(\mu_2) = \mathsf{var}(\mu_2') = \mathsf{var}(S)' :$
$(\mu_1{\oplus}\mu_2 \sim^{\mathcal{I}_1}_{\xi} \mu_1{\oplus}\mu_2 \ \wedge\ (S\,;S[\xi], \mu_1{\oplus}\mu_2) \rightsquigarrow^* (\surd, \mu_1'{\oplus}\mu_2'))$
$\Rightarrow \mu_1'{\oplus}\mu_2' \sim^{\mathcal{I}_2}_{\xi} \mu_1'{\oplus}\mu_2'$
iff {Logic}
$\forall \mu, \mu' :$
$(\exists \mu_1, \mu_2, \mu_1', \mu_2' : \mu = \mu_1{\oplus}\mu_2 \ \wedge\ \mu' = \mu_1'{\oplus}\mu_2' \wedge$
$\mathsf{var}(\mu_1) = \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge\ \mathsf{var}(\mu_2) = \mathsf{var}(\mu_2') = \mathsf{var}(S)')$
$\Rightarrow (\mu \sim^{\mathcal{I}_1}_{\xi} \mu \ \wedge\ (S\,;S[\xi], \mu) \rightsquigarrow^* (\surd, \mu'))$
$\Rightarrow \mu' \sim^{\mathcal{I}_2}_{\xi} \mu'$
iff {Existence of μ_1', μ_2' granted by Fact 1.1, and logic}
$\forall \mu, \mu' :$
$((\exists \mu_1, \mu_2 : \mu{=}\mu_1{\oplus}\mu_2 \wedge \mathsf{var}(\mu_1){=}\mathsf{var}(S) \wedge \mathsf{var}(\mu_2){=}\mathsf{var}(S)')$
$\quad \wedge\, \mu \sim^{\mathcal{I}_1}_{\xi} \mu \ \wedge\ (S\,;S[\xi], \mu) \rightsquigarrow^* (\surd, \mu'))$
$\Rightarrow \mu' \sim^{\mathcal{I}_2}_{\xi} \mu' \wedge$
$(\exists \mu_1', \mu_2' : \mu'{=}\mu_1'{\oplus}\mu_2' \wedge \mathsf{var}(\mu_1'){=}\mathsf{var}(S) \wedge \mathsf{var}(\mu_2'){=}\mathsf{var}(S)')$
iff {Characterisation of $\mathbf{I}_{sl}$}
$\forall \mu, \mu' : (\mu \models \mathbf{I}_{sl}(\mathcal{I}_1) \ \wedge\ (S\,;S[\xi], \mu) \rightsquigarrow^* (\surd, \mu'))$
$\qquad \Rightarrow \mu \models \mathbf{I}_{sl}(\mathcal{I}_2)$
iff {Separation Logic is sound and complete}
$\{\mathbf{I}_{sl}(\mathcal{I}_1)\}\ S\,;S[\xi]\ \{\mathbf{I}_{sl}(\mathcal{I}_2)\}$ is provable

D. Characterisation of security in CTL: proof for the termination sensitive case

S is TS $(\mathcal{I}_1, \mathcal{I}_2)$-secure
iff $\{$Cor. 1$\}$
$S \mathrel{\overset{\triangleright\, id,\mathcal{I}_1}{\approx_{id,\mathcal{I}_2}}} S[\xi]$
iff $\{$Def. 2$\}$
$\forall \mu_1, \mu_2, \mu_1' : \mathsf{var}(\mu_1) = \mathsf{var}(\mu_1') = \mathsf{var}(S) \wedge \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$\mu_1 \oplus \mu_2 \sim_\xi^{\mathcal{I}_1} \mu_1 \oplus \mu_2 \ \wedge \ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (S[\xi], \mu_1' \oplus \mu_2)$
$\Rightarrow \exists \mu_2' : \mathsf{var}(\mu_2') = \mathsf{var}(S)' :$
$\quad (S[\xi], \mu_1' \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'$
iff $\{$Satisfaction of $\mathsf{Ind}, \mathsf{end},$ and Fact 1.1$\}$
$\forall \mu_1, \mu_2, \mu_1' : \mathsf{var}(\mu_1) = \mathsf{var}(\mu_1') = \mathsf{var}(S) \wedge \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$\big((S \triangleright S[\xi], \mu_1 \oplus \mu_2) \models \mathsf{Ind}[\mathcal{I}_1] \wedge$
$\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (S[\xi], \mu_1' \oplus \mu_2) \big)$
$\Rightarrow \exists c : (S[\xi], \mu_1' \oplus \mu_2) \rightsquigarrow^* c \ \wedge \ c \models \mathsf{Ind}[\mathcal{I}_2] \wedge \mathsf{end}$
iff $\{$Semantics of EF and logic$\}$
$\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$(S \triangleright S[\xi], \mu_1 \oplus \mu_2) \models \mathsf{Ind}[\mathcal{I}_1]$
$\Rightarrow \forall \mu_1' : \mathsf{var}(\mu_1') = \mathsf{var}(S) :$
$\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (S[\xi], \mu_1' \oplus \mu_2)$
$\quad \Rightarrow (S[\xi], \mu_1' \oplus \mu_2) \models \mathsf{EF}(\mathsf{Ind}[\mathcal{I}_2] \wedge \mathsf{end})$
iff $\{$Satisfaction of $\mathsf{mid},$ Fact 1.1, and logic$\}$
$\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$(S \triangleright S[\xi], \mu_1 \oplus \mu_2) \models \mathsf{Ind}[\mathcal{I}_1]$
$\Rightarrow \forall c : (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* c$
$\qquad \Rightarrow c \models (\ \mathsf{mid} \rightarrow \mathsf{EF}(\mathsf{Ind}[\mathcal{I}_2] \wedge \mathsf{end})\ )$
iff $\{$Semantics of AG and $\rightarrow\}$
$\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$(S \triangleright S[\xi], \mu_1 \oplus \mu_2) \ \models$
$\qquad \mathsf{Ind}[\mathcal{I}_1] \rightarrow \mathsf{AG}(\ \mathsf{mid} \rightarrow \mathsf{EF}(\mathsf{Ind}[\mathcal{I}_2] \wedge \mathsf{end})\ )$
iff $\{$Notation considering that every μ can be separated $\}$
$S \triangleright S[\xi] \ \models \ \mathsf{Ind}[\mathcal{I}_1] \rightarrow \mathsf{AG}(\ \mathsf{mid} \rightarrow \mathsf{EF}(\mathsf{Ind}[\mathcal{I}_2] \wedge \mathsf{end})\ )$