Secure Information Flow by Self-Composition*

Gilles Barthe[1] Pedro R. D'Argenio[2]
Tamara Rezk (corresponding author) [3]

[3]IMDEA Software,
Campus Montegancedo,
28660-Boadilla del Monte, Madrid Spain,
Gilles.Barthe@imdea.org

[2]FaMAF, Universidad Nacional de Córdoba – CONICET,
Ciudad Universitaria,
5000 Córdoba, Argentina,
dargenio@famaf.unc.edu.ar

[3]INRIA, INDES Project,
2004, Route des Lucioles BP 93,
06902 Sophia Antipolis, France
TEL. +33 492387816 - FAX +33 492385060
Tamara.Rezk@sophia.inria.fr

June 29, 2010

Abstract

Information flow policies are confidentiality policies that control information leakage through program execution. A common means to enforce secure information flow is through information flow type systems. Although type systems are compositional and usually enjoy decidable type checking or inference, their extensibility is very poor: type systems need to be redefined and proven sound for each new single variation of security policy and programming language for which secure information flow verification is desired.

In contrast, program logics offer a general mechanism to enforce a variety of safety policies, and for this reason are favored in Proof Carrying Code, a promising security architecture for mobile code. However, the encoding of information flow policies in program logics is not

*The work is partially supported by ReSeCo. Gilles Barthe is partially supported by Parsec, Pedro R. D'Argenio is partially supported by PICT 26135.

straightforward, because they refer to a relation between two program executions.

The purpose of this paper is to investigate logical formulations of secure information flow based on the idea of self-composition, that reduces the problem of secure information flow of a program P to a safety property for a program $\hat{P}$ derived from P, by composing P with a renaming of itself. Self-composition enables the use of standard techniques for information flow policies verification, such as program logics and model checking, suitable in Proof Carrying Code infrastructures.

We illustrate the applicability of self-composition in several settings, including different security policies such as non-interference and controlled forms of declassification, and programming languages such as an imperative language with parallel composition, a non-deterministic language, and finally a language with shared mutable data structures.

1 Introduction

There is an increasing need to guarantee confidentiality of data in programming applications. In many cases, confidentiality is achieved through access control mechanisms that regulate access to sensitive data. However, these mechanisms do not guarantee that legitimately accessed data will not flow from authorized to unauthorized users. In order to achieve stronger confidentiality guarantees that account for the flow of information during program execution, an alternative is to use information flow policies, such as non-interference, a baseline information flow policy that guarantees the absence of information leakage. Informally, non-interference considers a partitioning of the program state into a public and a secret part, and requires that no information is leaked on the secret part of the state by observing the execution of the program. In its simplest instance, non-interference assumes that one can only observe the final value of the public state, and that (non)termination is not observable. Hence a program P is non-interfering if any two terminating executions of P starting from states that coincide on their public part yields final states that coincide on their public part; a more formal definition is given below. Non-interference and information flow policies have their roots in the works of Cohen [12, 13], Denning and Denning [17], and Goguen and Meseguer [22]; recently, they have attracted substantial interest within the language-based security, see [40] for a survey. Other forms of information flow policies, that are less strict and more appropriate than non-interference to be used in practice, include declassification policies, see [42] for a survey.

This paper is concerned with static enforcement of information flow poli-

cies in general. Currently, the prevailing means to enforce such policies is via information flow type systems [40]. Clearly, type systems are attractive because they support automated, compositional verification. However, type systems are inherently not extensible: every modification to the information flow policy or every new feature added to the programming language, requires a non-trivial extension of the type system and its soundness proof.

On the contrary, logical verification methods are flexible, and can be used to support several policies, without the need to prove soundness repeatedly. It is precisely for the ability of logic to support various policies that Proof Carrying Code [35, 36] relies on logical verification to validate mobile code on the consumer side. Typically, the consumer infrastructure consists of a verification condition generator, that operates on programs annotated with safety annotations, and a certificate checker, that verifies that the certificates validate the safety policy (from the soundness of the verification method, the certificate only has to show the validity proof obligations generated by the verification condition generator). On the other hand, certificate generation may be automated by certifying analyzers, that use type systems or static analyzes to generate the safety annotations, and generate automatically a proof of correctness of the program with respect to these annotations. In such cases, the certifying compiler crucially relies on the ability of the logic to express safety policies.

In order to extend the scope of Proof Carrying Code to expressive information flow policies, it is therefore important to understand how such policies can be encoded into traditional program logics. The encoding is not immediate, because information flow properties are not safety properties (as it is proved e.g. [33]), but rather properties of two or more execution traces.

The first encoding of information flow policies in Hoare logic is due to Andrews and Reitman [5, 17]. However, their encoding requires extending the set of axioms of Hoare logic in order to account for security properties. Therefore, such an encoding is impractical for our purposes, since we aim to capture information flow policies without changing the verification logic every time that the security property needs to be adapted. More recently, Darvas, Hähnle and Sands [16] have shown how dynamic logic may be used to verify non-interference and some declassification policies of (sequential) Java programs. As in Andrews and Reitman's encoding, non-interference of a program P is captured by a formula over P—in this case a formula in dynamic logic. However, this encoding relies on dynamic logic, rather than on more traditional Hoare logics, and its completeness has not been established.

In view of the difficulty of encoding an information flow policy for a

program P as a property of the same program, several works have taken a slightly different perspective on the problem, and reduced non-interference of a program P to a property about single program executions (universally quantified over all possible program inputs) of another program $\hat{P}$ constructed from P. This approach is taken for example in Pottier's work [37] on non-interference for the pi-calculus, where the non-interference of two processes $P1$ and $P2$ is reduced to a property about a single process P that captures the behaviours of $P1$ and $P2$ while keeping track of their shared sub-processes. The process P is written in an extension of the pi-calculus and allows for a simple proof of non-interference using standard subject reduction techniques. It is not clear for us whether this kind of technique easily extends to declassification policies.

In the conference article of this work [8], we called "self-composition" the reduction of information flow policies to a safety property: an information flow policy of a program P reduces to a property about single program executions (universally quantified over all possible program inputs) of the program $P; P'$, where P' is a renaming of P.

The reduction was further generalized by Terauchi and Aiken to the class of 2-safety properties [44] and by Clarkson and Schneider [11] to a wider class of properties. Thanks to self-composition, general-purpose logics such as Hoare-like logics or temporal logics, which provide a standard means to specify and verify safety properties of programs, can also be used to verify a wide range of information flow policies, and these policies can be handled in Proof Carrying Code infrastructures.

The objective of this work is to build upon self-composition to provide characterizations of information flow policies in programming and temporal logics. Our characterizations apply to many languages and different notions of security including some forms of declassification.

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 program variable in P is classified as either public or private and let $\vec{x}$ be the set of all public variables in P and $\vec{y}$ the set of all its private variables. Termination-insensitive non-interference for P may be cast as for all memories μ, μ', ν, ν':

$$[\langle P, \mu_1\rangle \Downarrow \nu_1 \ \wedge \ \langle P, \mu_2\rangle \Downarrow \nu_2 \ \wedge \ \mu_1 =_L \mu_2] \ \Rightarrow \nu_1 =_L \nu_2$$

where $=_L$ is the point-wise extension of equality on values to public parts of memories.

Let $[\vec{x}', \vec{y}'/\vec{x}, \vec{y}]$ be a renaming of the program variables $\vec{x}, \vec{y}$ of P with fresh variables $\vec{x'}, \vec{y'}$, and let P' be as program P but with its variables renamed with fresh names, that is $P[\vec{x}', \vec{y}'/\vec{x}, \vec{y}]$. Then, using $\uplus$ to denote the disjoint union of two memories, 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':

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

where $\circ$ represents function composition, and $=_{\vec{x}}$ is the point-wise extension of equality on values to the restriction of memories to $\vec{x}$. This new formulation reduces non-interference for program P to a property of every single execution of the program $P; P'$. Hence, 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 non-interference can be characterized as:

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

Let us now instantiate our characterization 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

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

which is easy to show using the rules of Hoare logic. By replacing the $=$-relation by other (partial) equivalence relation, we obtain characterizations of information flow policies that include some forms of declassification. More generally, this kind of characterization provides us with a means to resort to existing verification tools to prove, or disprove, information flow policies for 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 [38, 27] to provide a characterization of non-interference (see Section 8). Second, it can be extended to arbitrary relations between inputs and between outputs, as in e.g. [20]. 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 introduced by Sabelfeld and Myers [41].

Contributions. We conduct 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 secure information flow policies based on general purpose logics, 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 and other more general properties that can be defined as a relation between two executions of a program. A 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 8. This article is based on a paper by the authors, which was presented at the IEEE 17th Computer Security Foundations Workshop in June, 2004 [8]. The conference version of this work is subsumed by the present article in several ways:

- We provide a new characterization of termination sensitive secure information flow using a weakest precondition calculus, in the new Section 7.
- We review our formal framework correcting some mistakes introduced in [8]. In particular we review and formalize the assumptions on the framework of self-composition in the Preliminaries section as well as Fact 1.
- We discuss LTL logic characterization in Section 9 in greater detail and provide an example program with its characterization for termination sensitive and insensitive non-interference.
- We present complete proofs for the main theorems in the paper. We also provide proofs for programs in examples, that can be proved secure wrt information flow.
- We update the related work in order to account for the many works that appeared subsequent to the publication of [8].

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 replaced by variable y.

Assume given a set $\mathcal{M}$ of all memories, and let μ, μ', etc. range over $\mathcal{M}$. Further, with the purpose of defining security policies and properties on the programming languages considered, we assume two given functions: $\mathsf{var} : \mathcal{M} \to \mathsf{Var}$ and an abstraction function $\mathsf{v} : (\mathcal{M} \times \mathsf{Var}) \rightharpoonup \mathcal{V}$ with $\mathcal{V}$ being a set of values. $\mathsf{var}(\mu)$ returns the set of all variables whose values are stored in μ and we expect that if $x \in \mathsf{var}(\mu)$ then $\mathsf{v}(\mu, x)$ is defined. The value of $\mathsf{v}(\mu, x)$, depending on the language, may either represent just the value of the variable in memory μ, that is $\mu(x)$, or it may be the value represented by a data structure in the heap. (Notice that we will not use function v for expression evaluation semantics.) For example, if x is a pointer that contains an address in the heap that points to a linked list structure then $\mathsf{v}(\mu, x)$ returns the values in the list, abstracting from addresses used as links for the list (see Section 8 for a formal definition).

Our characterisations rely on the ability to update memories locally and to separate a memory into two disjoint pieces of memories. Both operations are specified as follows. First, if $\mu \in \mathcal{M}$, $x \in \mathsf{Var}$ and $d \in \mathcal{V}$, then $\mu[x \Mapsto d] \in \mathcal{M}$ is some memory s.t. for all $y \in \mathsf{Var}$ $\mathsf{v}(\mu[x \Mapsto d], y) =$ **if** $x = y$ **then** d **else** $\mathsf{v}(\mu, y)$. We remark that $\mu[x \Mapsto d]$ is one possible variation of μ —there might be more than one— such that $\mathsf{v}(\mu[x \Mapsto d], x) = d$. Thus, if x is a pointer and d is a list, $\mu[x \Mapsto d]$ is a modification of maybe many positions in the heap of μ and the assignation of the appropriate location (i.e. pointer value) to variable x. Second, if $\mu_1, \mu_2 \in \mathcal{M}$ are two memories that verify that $\mathsf{var}(\mu_1) \cap \mathsf{var}(\mu_2) = \emptyset$, we define $\mu_1 \oplus \mu_2 \in \mathcal{M}$ 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. We also require that $\mathsf{v}(\mu_1[x \Mapsto d] \oplus \mu_2, y) = \mathsf{v}(\mu_2, y)$ for all $x \in \mathsf{var}(\mu_1)$, $y \in \mathsf{var}(\mu_2)$, and $d \in \mathcal{V}$.

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 $\mu[x \Mapsto d](y) =$ **if** $x = y$ **then** d **else** $\mu(y)$.

The operational semantics of the programming language is given by the *transition system* $(\mathsf{Conf}, \rightsquigarrow)$ where $\mathsf{Conf} \subseteq \mathsf{Lang} \times \mathcal{M}$ 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 $c \rightsquigarrow c'$. (We assume standard expression evaluation semantics, and also assume configurations

consistency, that is if $(S,\mu) \in \mathsf{Conf}$ then $\mathsf{var}(S) \subseteq \mathsf{var}(\mu)$). 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 $(S,\mu)\perp$, 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

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

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, where memories are the functions of Example 1 and $\mu(e)$, the evaluation of a (boolean or arithmetic) expression e in memory μ, is recursively defined in the usual manner.

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

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

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

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

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

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

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

We now turn to state three basic assumptions which define the scope of application of our technique of self-composition. They impose some very general restrictions that can be seen as "healthiness conditions". Assumptions 1 and 3 are seemingly obvious and satisfied by most of the languages. 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. Depending on the definition of the abstraction function v, it may rule out programs.

Assumption 1. *Transitions preserve the set of variables of a program. Moreover, if the part of the memory that is affected by the program is separated from the rest, transitions do not affect the values of other variables than those appearing in the program.*

Formally, 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* $\exists \mu_1' : \mu' = \mu_1' \oplus \mu_2 \wedge \mathsf{var}(\mu_1') = \mathsf{var}(S)$. *In addition, if* $(S, \mu_1 \oplus \mu_2) \rightsquigarrow (S', \mu_1' \oplus \mu_2)$, *then for all* μ_3 *s.t.* $\mu_1 \oplus \mu_3$ *is defined,* $(S, \mu_1 \oplus \mu_3) \rightsquigarrow (S', \mu_1' \oplus \mu_3)$.

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 abstract value of its own variables.*

Formally, we assume that for all configurations (S, μ_1) *and* (S, μ_2) *such that* $\forall x \in \mathsf{var}(S) : \mathsf{v}(\mu_1, x) = \mathsf{v}(\mu_2, x)$ *then for all* (S', μ_1'), $(S, \mu_1) \rightsquigarrow^* (S', \mu_1') \Rightarrow \exists (S', \mu_2') : (S, \mu_2) \rightsquigarrow^* (S', \mu_2')$ *and* $\forall x \in \mathsf{var}(S) : \mathsf{v}(\mu_1', x) = \mathsf{v}(\mu_2', x)$.

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, program $\mathsf{if}\ (x{=}y) \rightarrow S\ []\ (x{\neq}y) \rightarrow S'\ \mathsf{fi}$ does not satisfy Assumption 2.

Assumption 3. *The operational semantics of the language* Lang *is independent of variable names. Formally, if* $y \notin \mathsf{var}(S)$ *and* $(S, \mu) \rightsquigarrow^* (S', \mu')$ *then* $(S[y/x], \mu[y \Mapsto \mathsf{v}(\mu, x)]) \rightsquigarrow^* (S'[y/x], \mu'[x \Mapsto d][y \Mapsto \mathsf{v}(\mu', x)])$ *for some* d.

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

The following facts follow from the assumptions above:

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 such that $\mathsf{var}(\mu_1) \cap \mathsf{var}(\mu_3) = \emptyset$.

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.

4. If $\forall x \in \mathsf{var}(S) : \mathsf{v}(\mu_1, x) = \mathsf{v}(\mu_2, x)$ and $(S, \mu_1)\bot$, then $(S, \mu_2)\bot$.

5. If $y \notin \mathsf{var}(S)$ and $(S, \mu)\bot$ then $(S[y/x], \mu[y \mapsto \mathsf{v}(\mu, x)])\bot$.

Items 1 and 2 follows from Assumption 1 while item 3 is a consequence of item 2. Items 4 and 5 are consequence of Assumptions 2 and 3, respectively. It is not difficult to verify that Par satisfies the three assumptions above and complies to Fact 1.

3 A Generalisation of Non-Interference

Let $\phi : \mathsf{Var} \rightharpoonup \mathsf{Var}$ 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 $\mu \sim^{\mathcal{I}}_{\phi} \mu'$, if $\langle(\mathsf{v}(\mu, x_1),..,\mathsf{v}(\mu, x_n)), (\mathsf{v}(\mu', \phi(x_1)),..,\mathsf{v}(\mu', \phi(x_n)))\rangle \in \mathcal{I}$, that is if the values of variables in memory μ, and the values of corresponding (according to ϕ) variables in memory μ' are related by relation $\mathcal{I}$.

Example 3. Let $L \subseteq \mathsf{Var}$ be the set of *low* (or *public*) variables of a program. Let $id_L : \mathsf{Var} \to \mathsf{Var}$ be the identity function on L and undefined otherwise. Then $\sim^{=}_{id_L}$ is the usual indistinguishability relation used to characterize 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. $\mathsf{avrg}([d_1, \ldots, d_N]) = \frac{d_1 + \cdots + d_N}{N}$. Let $id_H : \mathsf{Var} \to \mathsf{Var}$ be the identity function on H and undefined otherwise and let $\mathcal{A}$ be the relation including pairs of list of values $\langle [d_1 \ldots, d_N], [d'_1 \ldots, d'_N]\rangle$ such that $\mathsf{avrg}([d_1, \ldots, d_N]) = \mathsf{avrg}([d'_1, \ldots, d'_N])$. Then $\sim^{\mathcal{A}}_{id_H}$ cannot distinguish between memories μ and μ' which agree on the average value of the list to which p points, i.e. which verify $\mathsf{avrg}(\mathsf{v}(\mu, p)) = \mathsf{avrg}(\mathsf{v}(\mu', p))$.

At this point, function ϕ may be seen as redundant since it can always be encoded in $\mathcal{I}$. For instance, $\sim^{=}_{id_L}$ is equivalently defined by $\sim^{=_L}_{id}$, where id is the identity function and $=_L$ is the set

$$\{\langle(d_1, .., d_m, e_{m+1}, .., e_n), (d_1, .., d_m, e'_{m+1}, .., e'_n)\rangle \mid d_i, e_j, e'_j \in \mathcal{V}\}$$

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

provided $L = \{x_1, .., x_m\}$. The need for ϕ will become evident in Section 4 when security is defined using composition and variable renaming.

The next proposition follows from definition of $\sim$ and it claims that relation $\sim^{\mathcal{I}}_{\phi} \sim_{\phi}$ between memories depends only on the value of variables included in the domain of ϕ.

Proposition 1. *For all* μ_1, μ_2, μ_1'', μ_2'', $\mathcal{I}$, *and* $\phi : \mathsf{var}(\mu_1) \to \mathsf{var}(\mu_2)$, $\mu_1 \sim^{\mathcal{I}}_{\phi} \mu_2$ *iff* $\mu_1 \oplus \mu_1'' \sim^{\mathcal{I}}_{\phi} \mu_2 \oplus \mu_2''$.

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

Definition 1.

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

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

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

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

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 $S \approx^{id,\mathcal{I}}_{id,\mathcal{I}'} S$.

   (b) S is *termination insensitive (TI)* $(\mathcal{I}, \mathcal{I}')$*-secure* iff $S \approx^{id,\mathcal{I}}_{id,\mathcal{I}'} S$.

Informally, $S_1 \approx^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$ holds (read "S_1 is termination sensitive non-interferent with S_2") if for any two input indistinguishable memories, ones successful execution of S_1 from one of these memories, implies the existence of a successful execution of S_2 from the other memory, with both executions ending in output indistinguishable memories. $S_1 \approx^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$ (read "S_1 is termination insensitive non interferent with program S_2") is a weaker concept in the sense that S_2 might diverge. Finally, a program is (TS or TI) $(\mathcal{I}, \mathcal{I}')$-secure if, it is (TS or TI) non interferent with itself.

Traditional non-interference is characterized in our setting by $(=_L, =_L)$-security, with $=_L$ as defined above. It is not difficult to check that our definitions agree with those already defined in the literature (e.g. [22, 45, 43, 29]).

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

4 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 characterization of security:

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

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

Notice that this definition has the same shape as Definition 1. However, while $S_1 \overset{}{\approx}^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$ 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 ($S_1 \triangleright S_2$): the execution until the middle (that is, until S_2 is about to start) in the

antecedent of the implication, and the continuation of the execution until the end in the consequent.

The next theorem states 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).

Theorem 1. *Let S_1 and S_2 such that $\mathsf{var}(S_1) \cap \mathsf{var}(S_2) = \emptyset$ and let $\phi : \mathsf{var}(S_1) \to \mathsf{var}(S_2)$. Then*

(a) $S_1 \approx^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$ if and only if $S_1 \rhd \approx^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$, and

(b) $S_1 \approx^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$ if and only if $S_1 \rhd \lesssim^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$.

Proof. (a) *Termination sensitive case.* By Proposition 1 and commutativity of $\oplus$, we conclude that $\mu_1 \sim^{\mathcal{I}}_{\phi} \mu_2$ iff $\mu_1 \oplus \mu_2 \sim^{\mathcal{I}}_{\phi} \mu_1 \oplus \mu_2$. By Fact 1.2, we have that $(S_1, \mu_1) \rightsquigarrow^* (\surd, \mu_1')$ iff $(S_1, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2)$ and by definition of $\rhd$, $(S_1, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2)$ iff $(S_1 \rhd S_2, \mu_1 \oplus \mu_2) \rightsquigarrow^* (S_2, \mu_1' \oplus \mu_2)$.

Using similar arguments, we can conclude that

$$\begin{array}{rl} & \exists \mu_2' : (S_2, \mu_2) \rightsquigarrow^* (\surd, \mu_2') \ \wedge \ \mu_1' \sim^{\mathcal{I}'}_{\phi'} \mu_2' \\ \text{iff} & \exists \mu_2' : (S_2, \mu_1' \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim^{\mathcal{I}'}_{\phi'} \mu_1' \oplus \mu_2' \end{array} \tag{1}$$

from which (a) follows.

(b) *Termination insensitive case.* It follows by (1) and Fact 1.3 that

$$\begin{array}{rl} & \big(\exists \mu_2' : (S_2, \mu_2) \rightsquigarrow^* (\surd, \mu_2') \ \wedge \ \mu_1' \sim^{\mathcal{I}'}_{\phi'} \mu_2'\big) \vee (S_2, \mu_1')\bot \\ \text{iff} & \big(\exists \mu_2' : (S_2, \mu_1' \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim^{\mathcal{I}'}_{\phi'} \mu_1' \oplus \mu_2'\big) \vee (S_2, \mu_1' \oplus \mu_2)\bot \end{array}$$

Using equivalence of hypothesis shown in case (a), we can conclude. □

Programs sharing variable names are not handled by 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. This is stated in Theorem 2 below.

Before presenting Theorem 2, we prove an auxiliary lemma that is a weak version of the theorem in which only one variable is renamed. This lemma is used in the induction step of the proof of Theorem 2.

Lemma 1. *Let $y \notin (S_2)$, and let $[y/x] : \mathsf{var}(S_2) \to V$ be defined by $[y/x](z) =$ **if** $z \neq x$ **then** z **else** y. Then*

(a) $S_1 \approx^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$ *iff* $S_1 \approx^{[y/x]\phi,\mathcal{I}}_{[y/x]\phi',\mathcal{I}'} S_2[y/x]$, *and*

(b) $S_1 \approx^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$ *iff* $S_1 \approx^{[y/x]\phi,\mathcal{I}}_{[y/x]\phi',\mathcal{I}'} S_2[y/x]$.

where $S_2[y/x]$ *is program* S_2 *where variable* x *has been renamed by* y, *and* $[y/x]\phi$ *is a shorthand for* $[y/x] \circ \phi$.

Proof. **Case** (a).

Subcase ($\Rightarrow$). Let $\mu_1 \sim^{\mathcal{I}}_{[y/x]\phi} \mu_2$ and $(S_1, \mu_1) \rightsquigarrow^* (\surd, \mu_1')$. Notice that $\forall z \in \mathrm{dom}(\phi) : \mathsf{v}(\mu_2, [y/x]\phi(z)) = \mathsf{v}(\mu_2[x \Mapsto \mathsf{v}(\mu_2, y)], \phi(z))$, and hence $\mu_1 \sim^{\mathcal{I}}_{\phi} \mu_2[x \Mapsto \mathsf{v}(\mu_2, y)]$. As a consequence, since $S_1 \approx^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$, there is $\mu_2' \in \mathcal{M}$, such that,

$$(S_2, \mu_2[x \Mapsto \mathsf{v}(\mu_2, y)]) \rightsquigarrow^* (\surd, \mu_2') \text{ and } \mu_1' \sim^{\mathcal{I}'}_{\phi'} \mu_2' \tag{2}$$

By Assumption 3, there is some d such that

$$(S_2[y/x], \mu_2[x \Mapsto \mathsf{v}(\mu_2, y)][y \Mapsto \mathsf{v}(\mu_2[x \Mapsto \mathsf{v}(\mu_2, y)], x)]) \rightsquigarrow^* (\surd, \mu_2'[x \Mapsto d][y \Mapsto \mathsf{v}(\mu_2', x)]).$$

Since $x \notin \mathsf{var}(S_2[y/x])$, notice that for all $w \in \mathsf{var}(S_2[y/x])$,

$$v(\mu_2[x \Mapsto \mathsf{v}(\mu_2, y)][y \Mapsto \mathsf{v}(\mu_2[x \Mapsto \mathsf{v}(\mu_2, y)], x)], w) \;=\; v(\mu_2, w). \tag{3}$$

Hence, by Assumption 2,

$$(S_2[y/x], \mu_2) \rightsquigarrow^* (\surd, \mu_2'').$$

for some μ_2'' such that $\forall w \in \mathsf{var}(S_2[y/x]) : \mathsf{v}(\mu_2'[x \Mapsto d][y \Mapsto \mathsf{v}(\mu_2', x)], w) = \mathsf{v}(\mu_2'', w)$. Observe that $\forall z \in \mathrm{dom}(\phi') : \mathsf{v}(\mu_2', \phi'(z)) = \mathsf{v}(\mu_2'', [y/x]\phi'(z))$. In particular $y \notin \mathrm{dom}(\phi')$ and for z such that $\phi'(z) = x$,

$$\begin{aligned} \mathsf{v}(\mu_2'', [y/x]\phi'(z)) &= \mathsf{v}(\mu_2'', y) \\ &= \mathsf{v}(\mu_2'[x \Mapsto d][y \Mapsto \mathsf{v}(\mu_2', x)], y) \\ &= \mathsf{v}(\mu_2', x) \end{aligned}$$

As a consequence, and since $\mu_1' \sim^{\mathcal{I}'}_{\phi'} \mu_2'$, we finally have that, $\mu_1' \sim^{\mathcal{I}'}_{[y/x]\phi'} \mu_2''$.

Subcase ($\Leftarrow$). Clearly $x \notin \mathsf{var}(S_2[y/x])$ and, for all $z \in \mathrm{dom}(\phi)$ (recall $\mathrm{dom}(\phi) = \mathrm{dom}([y/x]\phi)$), $\phi(z) = \textbf{if } (([y/x]\phi)(z){=}y) \textbf{ then } x \textbf{ else } \phi(z)$ (and similarly for ϕ'). Using the previous case, where we take $S_2[y/x]$, $[y/x]\phi$ and $[y/x]\phi'$ instead of S_2, ϕ and ϕ', respectively, we have that $S_1 \approx^{[y/x]\phi,\mathcal{I}}_{[y/x]\phi',\mathcal{I}} S_2[y/x]$ implies $S_1 \approx^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2[y/x][x/y]$. Hence $S_1 \approx^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$.

Case (b). For the case of TI non-interference, we take over (2) and suppose, instead that $(S_2, \mu_2[x \Mapsto \mathsf{v}(\mu_2, y)])\bot$. By Fact 1.5, $(S_2[y/x], \mu_2[x \Mapsto \mathsf{v}(\mu_2, y)][y \Mapsto \mathsf{v}(\mu_2[x \Mapsto \mathsf{v}(\mu_2, y)], x)])\bot$. Taking into account equation (3) above, by Fact 1.4, $(S_2[y/x], \mu_2)\bot$, which, together with the previous case, proves ($\Rightarrow$). ($\Leftarrow$) follows reasoning as in subcase ($\Leftarrow$) of case (a). □

Theorem 2. *Let $\xi : \mathsf{var}(S_2) \to V$ be a bijective function on a set of variables V. Then*

(a) $S_1 \approxeq^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2$ iff $S_1 \approxeq^{\xi\circ\phi,\mathcal{I}}_{\xi\circ\phi',\mathcal{I}'} S_2[\xi]$, 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 have been renamed according to function ξ.

Proof. By induction on the number of variables x s.t. $\xi(x) \neq x$. Case $n = 0$ corresponds to the identity and it is trivial. Case $n \geq 1$ proceeds by induction using Lemma 1. In this case, we report only the proof of part (a). The induction proof of part (b) follows in the same manner. Let $x \in \mathsf{var}(S_2)$ and let ξ s.t. $\xi(x) = x$. Let y be a fresh variable not in the image of ξ. Notice that the number of variables z s.t. $[y/x]\xi(z) = z$ is exactly one more than those such that $\xi(z) = z$. Now we have

$$
\begin{aligned}
S_1 \approxeq^{[y/x]\xi\circ\phi,\mathcal{I}}_{[y/x]\xi\circ\phi',\mathcal{I}'} S_2[[y/x]\xi] \text{ iff } & S_1 \approxeq^{[y/x](\xi\circ\phi),\mathcal{I}}_{[y/x](\xi\circ\phi'),\mathcal{I}'} S_2[\xi][y/x] && \text{(By calculations)}\\
\text{iff } & S_1 \approxeq^{\xi\circ\phi,\mathcal{I}}_{\xi\circ\phi',\mathcal{I}'} S_2[\xi] && \text{(By applying Lemma 1)}\\
\text{iff } & S_1 \approxeq^{\phi,\mathcal{I}}_{\phi',\mathcal{I}'} S_2 && \text{(By induction hypothesis)}
\end{aligned}
$$

□

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

Corollary 1. *Let $\xi : \mathsf{var}(S) \to \mathsf{Var}$. Define $\mathsf{var}(S)' = \{\xi(x) \mid x \in \mathsf{var}(S)\}$, so that $\mathsf{var}(S) \cap \mathsf{var}(S)' = \emptyset$ and $x \mapsto \xi(x)$ is a bijection from $\mathsf{var}\, S$ to $\mathsf{var}(S)'$. Then, the following statements are equivalent*

1. *S is TS (resp. TI) $(\mathcal{I}, \mathcal{I}')$-secure.*
2. *$S \approxeq^{\xi,\mathcal{I}}_{\xi,\mathcal{I}'} S[\xi]$ (resp. $S \approx^{\xi,\mathcal{I}}_{\xi,\mathcal{I}'} S[\xi]$)*
3. *$S \rhd \approxeq^{\xi,\mathcal{I}}_{\xi,\mathcal{I}'} S[\xi]$ (resp. $S \rhd \approx^{\xi,\mathcal{I}}_{\xi,\mathcal{I}'} S[\xi]$)*

Corollary 1 allows to check whether a program S is secure by analyzing single executions of the program $S \rhd S[\xi]$. But this is what verification logics are used for. We characterize $(\mathcal{I}, \mathcal{I}')$-security in some of such logics.

5 Deterministic Programs

Simpler definitions for non-interference can be obtained if the program S under study is deterministic. We say that a program S is *deterministic* if for every memory μ and configurations c, c_1', and c_2', if $(S,\mu) \rightsquigarrow^* c$, $c \rightsquigarrow c_1'$ and $c \rightsquigarrow c_2'$, then $c_1' = c_2'$. From here, it should not be difficult to verify that if S is deterministic, for all μ, either $(S,\mu)\bot$ or there is a unique memory μ' such that $(S,\mu) \rightsquigarrow^* (\surd, \mu')$.

Assuming determinism, the definition of security is simpler than Definition 2 since we do not need to reference to intermediate points in the program and instead consider only complete executions. This allows to check security by simply analyzing the I/O behaviour of the self-composed program $S; S[\xi]$. This intuition is captured in the following theorem.

Theorem 3. *Let S be a deterministic program and $\xi : \mathsf{var}(S) \to \mathsf{Var}$ and $\mathsf{var}(S)'$ as in Corollary 1.*

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

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

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

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

The proof of Theorem 3 can be found in Appendix A.

This theorem can be extended to programs that are deterministic only for the observed value of the variables (and not necessarily in the internal representation). We say that a program S is *observationally deterministic* if for all programs S', S_1', and S_2' and memories μ, μ_1, μ_2, μ_1' and μ_2' with $\mathsf{v}(\mu_1, x) = \mathsf{v}(\mu_2, x)$ for all $x \in \mathsf{Var}$, if $(S,\mu) \rightsquigarrow^* (S', \mu_1)$, $(S,\mu) \rightsquigarrow^* (S', \mu_2)$, $(S', \mu_1) \rightsquigarrow (S_1', \mu_1')$, and $(S', \mu_2) \rightsquigarrow (S_2', \mu_2')$, then $S_1' = S_2'$ and $\mathsf{v}(\mu_1', x) = \mathsf{v}(\mu_2', x)$ for all $x \in \mathsf{Var}$. Notice, in particular, that if $(S,\mu) \rightsquigarrow (S_1, \mu_1)$ and $(S,\mu) \rightsquigarrow (S_2, \mu_2)$ then $S_1' = S_2'$ and $\mathsf{v}(\mu_1', x) = \mathsf{v}(\mu_2', x)$ for all $x \in \mathsf{Var}$. Moreover, it can be proved that either $(S,\mu)\bot$ or for all $\mu_1, \mu_2 \in \{\mu' \mid (S,\mu) \rightsquigarrow^* (\surd, \mu')\}$, $\mathsf{v}(\mu_1, x) = \mathsf{v}(\mu_2, x)$ for every $x \in \mathsf{Var}$.

The proof of the next theorem follows closely the proof of Theorem 3 (see Appendix A for the proof).

Theorem 4. *Let S be an observationally deterministic program and let $\xi : \mathsf{var}(S) \to \mathsf{Var}$ and $\mathsf{var}(S)'$ as in Corollary 1. Then equivalences 1. and 2. in Theorem 3 hold.*

The following example shows that the alternative definition given by Theorem 3 for deterministic program does not extend to non-deterministic programs in general.

Example 4. Recall the non-deterministic language Par. The non-deterministic program $\mathsf{if}\ x = 1 \to x := 2\ [\!]\ x = 1 \to x := 1\ \mathsf{fi}$, where x is public, is TI and TS $(=_L, =_L)$-secure according to Definition 2. However it does not satisfy conditions of Theorem 3 since starting from indistinguishable states with $x = 1$, the self-composed program will not always terminate in states where x has the same value.

Theorem 3 can be further enhanced for languages featuring simple functional memories like the one defined in Example 1 and that will be central in the next two sections. Notice that Theorem 3 requires that memory should be separable by operation $\oplus$ (μ is separable by $\oplus$ if there are μ_1 and μ_2 such that $\mu = \mu_1 \oplus \mu_2$). Functions can *always* be separated (this is not the case with more complex memories like those in Section 8). 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 $\xi : \mathsf{var}(S) \to \mathsf{Var}$ and $\mathsf{var}(S)'$ as in Corollary 1.*

1. *S is TS $(\mathcal{I}_1, \mathcal{I}_2)$-secure if and only if*

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

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

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

6 Hoare Logic

In this section we use the results of self composition to characterize $(\mathcal{I}_1, \mathcal{I}_2)$-security in Hoare logic.

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

Let P and Q be first order predicates and S a While program. A Hoare triple [24] $\{P\}\,S\,\{Q\}$ 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:

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

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

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

Hoare logic is sound and (relatively) complete w.r.t. operational semantics [14]. 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 first order predicate representing the indistinguishability criterion $\mathcal{I}$ on the values of the program, that is,

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

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 assertion language embedded in 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 $\bigwedge_{x \in L} x = x'$.

Proposition 2. *Termination insensitive $(\mathcal{I}_1, \mathcal{I}_2)$-security can be characterized in Hoare logic as follows:*

$$S \textit{ is TI } (\mathcal{I}_1, \mathcal{I}_2)\textit{-secure iff } \{\mathbf{I}(\mathcal{I}_1)\}\,S\,;\,S[\xi]\,\{\mathbf{I}(\mathcal{I}_2)\} \textit{ is provable.}$$

Proof. We remark that the language above has semantics on memories like in Example 1. Moreover the language is deterministic and we take the sequential composition to be operator $\triangleright$. Therefore, we are under the conditions of Corollary 2, which is central to this proof.

S is TI $(\mathcal{I}_1, \mathcal{I}_2)$-secure
iff {Corollary 2.2}
$\forall \mu, \mu' : \big(\ \mu \sim_\xi^{\mathcal{I}_1} \mu \ \wedge\ (S\,;\,S[\xi], \mu) \rightsquigarrow^* (\surd, \mu')\ \big) \Rightarrow \mu' \sim_\xi^{\mathcal{I}_2} \mu'$
iff {Def. of $\mathbf{I}$}
$\forall \mu, \mu' : \big(\ \mu \models \mathbf{I}(\mathcal{I}_1) \ \wedge\ (S\,;\,S[\xi], \mu) \rightsquigarrow^* (\surd, \mu')\ \big) \Rightarrow \mu' \models \mathbf{I}(\mathcal{I}_2)$
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 5. 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 noninterferent. Indistinguishability in this case is characterized by predicate $\mathbf{I}(=_{\{x_l\}}) \equiv (x_l = x_l')$. The proof is given in Figure 1(a).

The generality of our definition is useful for providing a characterization of some forms of controlled declassification. Declassification allows to leak some confidential information without being too revealing. A semantic characterization of this kind of properties has been given in [41] and coined *delimited release*. 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), see Example 8. Another typical example is given by access control procedures such as the following instance.

Example 6. This example —the PIN access control— deals with declassification. In the program

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

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 $\sim_{id}^{\mathcal{I}}$ 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 outlined in Figure 1(b).

7 Weakest precondition

Partial correctness is not enough to formulate a characterization of termination sensitive $(\mathcal{I}, \mathcal{I}')$-security for deterministic programs, where one needs to ensure that if S terminates for some memory μ_1, $S[\xi]$ terminates for indistinguishable memory μ_2 (Theorem 3). However by using total correctness specifications and self-composition, it is possible to specify TS security using the weakest conservative precondition [18] (wp).

$\{x_l = x_l'\}$
$\{x_l + y_h - y_h = x_l'\}$
$x_l := x_l + y_h$;
$\{x_l - y_h = x_l'\}$
$x_l := x_l - y_h$;
$\{x_l = x_l'\}$
$\{x_l = x_l' + y_h' - y_h'\}$
$x_l' := x_l' + y_h'$;
$\{x_l = x_l' - y_h'\}$
$x_l' := x_l' - y_h'$
$\{x_l = x_l'\}$

(a)

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

(b)

Figure 1: Security proof in Hoare logic

Given two predicates P, Q and a program S, predicate transformer wp is sound and complete in the following sense:

$$P \Rightarrow wp(S,Q) \text{ iff } \forall \mu : \ \mu \models P \Rightarrow \ \exists \mu' : \ (S,\mu) \rightsquigarrow^* (\surd, \mu') \ \wedge \mu' \models Q \qquad (4)$$

In particular,

$$\mu \models wp(S, \mathsf{true}) \text{ iff } \exists \mu' : \ (S,\mu) \rightsquigarrow^* (\surd, \mu') \qquad (5)$$

Therefore, $wp(S, \text{true})$ characterizes the set of memories in which the execution of S terminates.

The equations for the calculus of $wp(S,Q)$ are the following:

$$\begin{array}{lcl}
wp(x := e, Q) & = & Q[e/x] \\
wp(\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}, Q) & = & b \Rightarrow wp(S_1, Q) \wedge \neg b \Rightarrow wp(S_2, Q) \\
wp(S_1\ ;\ S_2, Q) & = & wp(S_1, wp(S_2, Q)) \\
wp(\mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}, Q) & = & \exists k : k \geq 0 : H_k(Q)
\end{array}$$

where

$$H_0(Q) = \neg b \wedge Q$$
$$H_{k+1}(Q) = (b \wedge wp(S, H_k(Q))) \vee H_0(Q)$$

Proposition 3. *Termination sensitive $(\mathcal{I}_1, \mathcal{I}_2)$-security can be characterized using wp as follows:*

$$S \textit{ is TS } (\mathcal{I}_1, \mathcal{I}_2)\textit{-secure iff } \mathbf{I}(\mathcal{I}_1) \wedge wp(S, \mathsf{true}) \Rightarrow wp(S\,;S[\xi], \mathbf{I}(\mathcal{I}_2)).$$

Proof.

S is TS $(\mathcal{I}_1, \mathcal{I}_2)$-secure
iff {Corollary 2.1}
$\forall \mu : \big(\ (\ \mu \sim_{\xi}^{\mathcal{I}_1} \mu\ \wedge\ \exists \mu' :\ (S,\mu) \rightsquigarrow^* (\surd, \mu')\ )$
$\qquad \Rightarrow (\ \exists \mu'' :\ (S; S[\xi], \mu) \rightsquigarrow^* (\surd, \mu'')\ \wedge\ \mu'' \sim_{\xi}^{\mathcal{I}_2} \mu''\ )\ \big)$
iff {Def. of $\mathbf{I}$}
$\forall \mu : \big(\ (\ \mu \models \mathbf{I}(\mathcal{I}_1)\ \wedge\ \exists \mu' : (S,\mu) \rightsquigarrow^* (\surd, \mu')\ )$
$\qquad \Rightarrow (\ \exists \mu'' :\ (S; S[\xi], \mu) \rightsquigarrow^* (\surd, \mu'')\ \wedge\ \mu'' \models \mathbf{I}(\mathcal{I}_2)\ )\ \big)$
iff {By (5)}
$\forall \mu : \big(\ (\ \mu \models \mathbf{I}(\mathcal{I}_1)\ \wedge\ \mu \models wp(S, \mathsf{true})\ )$
$\qquad \Rightarrow (\ \exists \mu'' :\ (S; S[\xi]), \mu) \rightsquigarrow^* (\surd, \mu'')\ \wedge\ \mu'' \models \mathbf{I}(\mathcal{I}_2)\ )\ \big)$
iff {By (4)}
$(\ \mathbf{I}(\mathcal{I}_1)\ \wedge\ wp(S, \mathsf{true})\ ) \Rightarrow wp(S; S[\xi], \mathbf{I}(\mathcal{I}_2))$

□

The following example shows a program that is termination insensitive secure but it is not termination sensitive secure. Thus, the program can be proved secure using the characterization of Hoare logic with partial correctness, but verification fail using characterization of wp.

Example 7. Consider program S where y is a high variable:

```
while y < 3 do
  if y < 1 then
    y := y − 1
  else
    y := y + 1
  fi
od
```

Since there are no low variables in S, indistinguishability criteria are trivially true. Using equations for wp, we calculate $wp(S, \mathsf{true})$:

$$\exists k : k \geq 0 : y \geq 3 \vee (y \leq 3 \wedge y \geq 1 \wedge y \geq 3 - k)$$

what is equivalent to $y \geq 1$.

If we calculate $wp(S\,;S[y'/y], \mathsf{true})$ we obtain $y \geq 1 \wedge y' \geq 1$, which is not implied by $wp(S, \mathsf{true})$ that is $y \geq 1$. Hence, program S (that is trivially TI secure since there are no low variables) is not TS secure.

Hoare-logic with total correctness and wp relates by $P \Rightarrow wp(S, Q)$ iff $[\,P\,]\,S\,[\,Q\,]$, where $[\,P\,]\,S\,[\,Q\,]$ denotes the Hoare triple for total correctness.

Following Proposition 3, a first attempt to characterize $(\mathcal{I}_1, \mathcal{I}_2)$-security using Hoare-logic with total correctness yields $[\,\mathbf{I}(\mathcal{I}_1) \wedge wp(S, \mathsf{true})\,]\, S\, ;\, S[\xi]\, [\,\mathbf{I}(\mathcal{I}_2)\,]$.

However this characterization is impure in the sense that it mixes the calculus of weakest precondition and Hoare logic triples for total correctness. Since $wp(S, \mathsf{true})$ is the weakest predicate P such that $[\,P\,]\, S\, [\,\mathsf{true}\,]$, it turns out that the characterization using Hoare-logic with total correctness is, not only impossible in its pure form, but also requires a second order quantification. In fact, the characterization should be written as follows:

$$\forall P : [\,P\,]\, S\, [\,\mathsf{true}\,] : [\,\mathbf{I}(\mathcal{I}_1) \wedge P\,]\, S\, ;\, S[\xi]\, [\,\mathbf{I}(\mathcal{I}_2)\,]$$

This justifies our choice to use wp rather than Hoare logic for total correctness to characterize $(\mathcal{I}_1, \mathcal{I}_2)$-security.

8 Separation Logic

Separation logic is an extension of Hoare logic to reason about shared mutable data structures [27, 39]. While^p extends the While language with the following commands:

$$S ::= \ \cdots \mid x := e.i \mid x.i := e \mid x := \mathsf{cons}(e_1, e_2) \mid \mathsf{dispose}(e) \tag{6}$$

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, and $\mathsf{dispose}(e)$ deallocates a cell from the heap. Furthermore $e.i$ 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 $\mathcal{V} = \mathbb{Z} \cup \mathsf{Loc}$, then $\mathcal{S} = \mathsf{Var} \to \mathcal{V}$ is the set of *stores* and $\mathcal{H} = \mathsf{Loc} - \{\mathsf{nil}\} \to (\mathcal{V} \times \mathsf{Loc})$ is the set of *heaps*. As a consequence variables can have type $\mathbb{Z}$ or type Loc. Finally $\mathcal{M} = \mathcal{S} \times \mathcal{H}$.

Separation logic requires additional predicates to make assertions about pointers. In addition to formulas of the classical predicate calculus, the logic has the following forms of assertions: $e \mapsto (e_1, e_2)$ which holds in a singleton heap with location satisfying e and the cell values satisfying e_1 and e_2 respectively; emp 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 also the separating implication $P \mathrel{-\!\!*} Q$, see [27, 38]. The meaning of an assertion depends upon both the store and the heap:

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

where $s(e)$ is the standard meaning of an expression given the store s. 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 $e \mapsto (\_, e_2)$ abbreviate "$\exists e' : e \mapsto (e', e_2)$ and variables occurring in e' are not free in e neither e_2", then

$$\{e \mapsto (\_, e_2)\}\, e.1 := e_1\, \{e \mapsto (e_1, e_2)\}$$

- If x does not occur in e_1 or in e_2 then

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

- If x, x' and x'' are different and x does not occur in e neither e_2, then

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

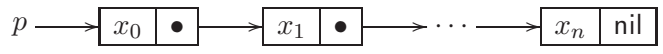
- Finally,

$$\{\exists e_1, e_2 : e \mapsto (e_1, e_2)\}\, \mathsf{dispose}(e)\, \{\mathsf{emp}\}$$

Using separation logic we can define inductive predicates to make reference to structures in the heap (see [38, 27]). For simplicity we only consider predicate list which is defined by:

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

For instance, predicate $\mathsf{list}.[x_0, \ldots, x_n].p$ is valid only in the heap represented below:



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.,

$$\overline{\mathsf{v}}(h,l) = \textbf{if } l{=}\mathsf{nil} \textbf{ then } [\,] \textbf{ else } \mathrm{fst}(h(l)){:}\overline{\mathsf{v}}(h{-}\{(l,h(l))\},\ \mathrm{snd}(h(l)))$$

and

$$(s_1,h_1)\oplus(s_2,h_2) \;=\; (s_1{\oplus}s_2, h_1{\oplus}h_2) \tag{7}$$

is defined only if all locations reachable from store s_i are defined in the heap h_i, $i = 1,2$. Formally, a location l' is reached from a location l in a heap h if $l' \in \mathrm{reach}(l,h) = \{(\mathrm{snd}\circ h)^k(l) \mid k \geq 0\}$ Then, the *set of all locations reachable from store s in h* is defined by $\mathrm{reach}(s,h) = \bigcup\{\mathrm{reach}(l,h) \mid l \in \mathrm{ran}(s) \cap \mathsf{Loc} - \{\mathsf{nil}\}\}$. Then, (7) is defined if $\mathrm{reach}(s_i,h_i) \subseteq \mathrm{dom}(h_i)$ for all $i = 1,2$. 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).

Now that v and $\overline{\mathsf{v}}$ are defined, notice that

$$(s,h) \models \mathsf{list}.xs.x * \mathsf{true} \;\;\text{iff}\;\; \overline{\mathsf{v}}(h,s(x)) = xs \;\;\text{iff}\;\; \mathsf{v}((s,h),x) = xs \tag{8}$$

Let $\{x_1,..,x_n\}$ be all variables in S that have type Loc (the pointer variables) and $\{y_1,..,y_m\}$ all variables in S of type $\mathbb{Z}$ (the integer variables). Let $\vec{x} = (x_1,..,x_n)$ and $\vec{x}' = (x_1',..,x_n')$ and similarly for $\vec{y}$ and $\vec{y}'$. Denote $\vec{xs} = (xs_1,..,xs_n)$ and $\vec{xs}' = (xs_1',..,xs_n')$. Fix this notation for the rest of this section.

Let $\mathcal{I}$ be the indistinguishability criterion. Notice that, in this setting, $\mathcal{I}$ deals with values in $\mathbb{Z}$ and also with *lists*, which are the interpretation of pointer variables. Assume there are m integer variables and n pointer variables, then $\mathcal{I} \subseteq (\mathbb{Z}^m \times \mathbb{L}^n) \times (\mathbb{Z}^m \times \mathbb{L}^n)$ with $\mathbb{L}$ being the set of all possible lists.

Let $\mathbf{I}_{sl}(\mathcal{I})$ be predicate

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

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 the following holds:

$$\begin{array}{lll} \mu \models \mathbf{I}_v(\mathsf{v}(\mu, \vec{x}), \mathsf{v}(\mu, \vec{x'}), \mathcal{I}) & \text{iff} & (\langle \mathsf{v}(\mu, \vec{y}), \mathsf{v}(\mu, \vec{x}) \rangle, \langle \mathsf{v}(\mu, \vec{y'}), \mathsf{v}(\mu, \vec{x'}) \rangle) \in \mathcal{I} \\ & \text{iff} & \mu \sim_{\xi}^{\mathcal{I}} \mu. \end{array}$$

Yet, μ needs to be separable so that we are in the setting of Theorem 4. That is why we also require that

$$\mu \models \exists \vec{xs}, \vec{xs'} : \Big(\Big(\bigwedge_{1 \leq i \leq n} (\mathsf{list}.xs_i.x_i * \mathsf{true}) \Big) * \Big(\bigwedge_{1 \leq i \leq n} (\mathsf{list}.xs'_i.x'_i * \mathsf{true}) \Big) \Big).$$

Therefore, $\mathbf{I}_{sl}(\mathcal{I})$ has two parts: the first part states the separation of the heap identifying the list values represented by the pointer variables, and the second one, the proper indistinguishability of the values (including also the values of the integer variables).

Separation logic is (relatively) complete for the language we are using [27]. As a consequence, security in separation logic can be completely characterized as follows:

Proposition 4. *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.*

Before continuing with the proof we state the following properties:

Property 1. Every While^p program is observationally deterministic.

Notice that, even if we assume that the heap allocator is non-deterministic (the heap allocator is used for the semantics of cons to create a fresh address in the heap), the semantics of the While^p language is still deterministic in the sense that the same program with the same inputs produces the same outputs. This is due to the fact that the language disallows comparisons between addresses in the heap (tests on pointer values are disallowed).

Notice that if tests on pointers were allowed (that is, Property 1 would not be valid), new leaks can arise throughout address values. Consider for example program $p_l := \mathsf{cons}(1, \mathsf{nil})\ ;\ q_l := \mathsf{cons}(1, \mathsf{nil})\ ;\ \mathsf{if}\ p_l < q_l\ \mathsf{then}\ x_l := 1\ \mathsf{else}\ x_l := 2\ \mathsf{fi}$ with p_l, q_l, x_l being public variables. Assume that the allocator depends on secret information being allocated before this public command. Then this program is insecure, since at the end of the program,

depending on the location assigned by the allocator to p_l and q_l, value of x_l will be 0 or 1, revealing whether secret information has been allocated before.

Properties 1 follows by structural induction using the operational rules defined in [27]. Using previous observations, we finally proceed to prove correctness and completeness of the characterization in Separation Logic.

Proof. We prove first that $\mathbf{I}_{sl}(\mathcal{I})$ characterizes 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, observe that $\overline{\mathsf{v}}(h, l)$ is defined 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 $\mathrm{dom}(s) = \mathsf{var}(S) \cup \mathsf{var}(S)'\}$

$$\begin{aligned}\exists s_1, s_2, h_1, h_2 : h = h_1 \oplus h_2 \wedge s = s_1 \oplus s_2 \\ \wedge\ \mathrm{dom}(s_1) = \mathsf{var}(S) \wedge \mathrm{dom}(s_2) = \mathsf{var}(S)' \\ \wedge\ \overline{\mathsf{v}}(h_1, s(\vec{x})) \text{ and } \overline{\mathsf{v}}(h_2, s(\vec{x}')) \text{ are defined}\end{aligned}$$

iff {Observation above}

$$\begin{aligned}\exists s_1, s_2, h_1, h_2 : h = h_1 \oplus h_2 \wedge s = s_1 \oplus s_2 \\ \wedge\ \mathrm{dom}(s_1) = \mathsf{var}(S) \wedge \mathrm{dom}(s_2) = \mathsf{var}(S)' \\ \wedge\ \mathrm{reach}(s_i, h_i) \subseteq \mathrm{dom}(h_i) \text{ for } i \in \{1, 2\}\end{aligned}$$

iff {Def. of $\oplus$ and $\mathsf{var}(s_i, h_i) = \mathrm{dom}(s_i)$}

$$\begin{aligned}\exists s_1, s_2, h_1, h_2 : (s, h) = (s_1, h_1) \oplus (s_2, h_2) \\ \wedge\ \mathsf{var}(s_1, h_1) = \mathsf{var}(S) \wedge \mathsf{var}(s_2, h_2) = \mathsf{var}(S)'\end{aligned} \tag{9}$$

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

$(s, h) \models \mathbf{I}_{sl}(\mathcal{I})$

iff {unfolding of $\mathbf{I}_{sl}(\mathcal{I})$}

$$\begin{aligned}(s, h) \models \exists \vec{ds}, \vec{ds}' : \left(\textstyle\bigwedge_{1 \le i \le n} (\mathsf{list}.ds_i.x_i * \mathsf{true})\right) * \left(\textstyle\bigwedge_{1 \le i \le n} \mathsf{list}.ds'_i.x'_i * \mathsf{true})\right) \\ \wedge\ \mathbf{I}_v(\vec{xs}, \vec{xs}', \mathcal{I})\end{aligned}$$

iff {By semantics (equation 8) }

$$\begin{aligned}\exists \vec{ds}, \vec{ds}' : \exists h_1, h_2 : h = h_1 \oplus h_2 : \\ (\forall i : 1 \le i \le n : ds_i = \overline{\mathsf{v}}(h_1, s(x_i)) \wedge ds'_i = \overline{\mathsf{v}}(h_2, s(x'_i))) \\ \wedge\ (s, h) \models \mathbf{I}_v(\vec{ds}, \vec{ds}', \mathcal{I})\end{aligned}$$

iff {Def. of $\mathbf{I}_v$ and equality on vectors}

$\exists \vec{ds}, \vec{ds}' : \exists h_1, h_2 : h = h_1 \oplus h_2 : \vec{ds} = \mathsf{v}((s, h_1), \vec{x}) \ \wedge \ \vec{ds}' = \mathsf{v}((s, h_2'), \vec{x}')$
$\qquad \wedge \ (\langle \mathsf{v}(\mu, \vec{y}), \vec{ds} \rangle, \langle \mathsf{v}(\mu, \vec{y}'), \vec{ds}' \rangle) \in \mathcal{I}$

iff $\{ \ \overline{\mathsf{v}}(h_1, s(x_i)) = \mathsf{v}((s, h), x_i) = ds_i$ and
$\overline{\mathsf{v}}(h_2, s(x_i')) = \mathsf{v}((s, h), x_i') = ds_i', \quad \text{for } 1 \leq i \leq n \}$

$\exists \vec{ds}, \vec{ds}' : \exists h_1, h_2 : h = h_1 \oplus h_2 : \vec{ds} = \mathsf{v}((s, h_1), \vec{x}) \ \wedge \ \vec{ds}' = \mathsf{v}((s, h_2'), \vec{x}')$

$\qquad \wedge \ (\langle \mathsf{v}(\mu, \vec{y}), \mathsf{v}(\mu, \vec{x}) \rangle, \langle \mathsf{v}(\mu, \vec{y}'), \mathsf{v}(\mu, \vec{x}') \rangle) \in \mathcal{I}$

iff $\{\exists v : f(z) = v$ iff $f(z)$ is defined, and Def. of $\sim_\xi^{\mathcal{I}}\}$

$\exists h_1, h_2 : h = h_1 \oplus h_2 : \overline{\mathsf{v}}(h_1, s(\vec{x}))$ and $\overline{\mathsf{v}}(h_2, s(\vec{x}'))$ are defined $\ \wedge \ (s, h) \sim_\xi^{\mathcal{I}} (s, h)$

iff { Remark (9) }

$$\exists s_1, s_2, h_1, h_2 : (s, h) = (s_1, h_1) \oplus (s_2, h_2) \wedge \mathsf{var}(s_1, h_1) = \mathsf{var}(S) \wedge \mathsf{var}(s_2, h_2) = \mathsf{var}(S)' \wedge (s, h) \sim_\xi^{\mathcal{I}} (s, h) \tag{10}$$

S is TI $(\mathcal{I}_1, \mathcal{I}_2)$-secure

iff {Property 1 and Theorem 4}

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

iff {Logic}

$\forall \mu, \mu' : \big(\ \exists \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' \ \wedge \ \mu = \mu_1 \oplus \mu_2 \big) \wedge$
$\big(\ \exists \mu_1', \mu_2' : \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2') = \mathsf{var}(S)' \ \wedge \ \mu' = \mu_1' \oplus \mu_2' \ \big) :$
$\big(\ \mu \sim_\xi^{\mathcal{I}_1} \mu \ \wedge \ (S; S[\xi], \mu) \rightsquigarrow^* (\surd, \mu') \ \big) \Rightarrow \mu' \sim_\xi^{\mathcal{I}_2} \mu'$

iff {Logic and Remark (10) }

$\forall \mu, \mu' : \big(\ \mu \models \mathbf{I}_{sl}(\mathcal{I}_1) \ \wedge \ (S; S[\xi], \mu) \rightsquigarrow^* (\surd, \mu') \ \big) \Rightarrow \mu' \models \mathbf{I}_{sl}(\mathcal{I}_2)$

iff {Separation Logic is sound and complete}

$\{\mathbf{I}_{sl}(\mathcal{I}_1)\}\, S\, ;\, S[\xi]\, \{\mathbf{I}_{sl}(\mathcal{I}_2)\}$ is provable

□

Example 8. The following program receives a list *lsalaries* with employees salaries and returns in a_l the average of the salaries. We use projections *.salary* and *.next* as syntactic sugar for projections .1 and .2 on lists.

$p := lsalaries\, ;\, s := 0\, ;\, n := 0\, ;$
while $p \neq \mathsf{nil}$ **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 AV_SAL (for "AVerage SALary").

The security requirement is to reveal only the average of the employee salaries without revealing any information about individual salaries. In this sense, the only public variable in AV_SAL is a_l, which is intended to store the average salary resulting from the calculation. If $\mathcal{A}$ is defined as in Example 3 (except that the length of the list of salaries is not fixed as in Example 3), we expect AV_SAL 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:

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

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

Here, we use notation $|ps|$ for the length of list ps, and $\sum ps$ for the sum of all numbers in ps with $\sum[\,] = 0$. We suppose that the heap contains exactly the lists pointed by *lsalaries* and *lsalaries*$'$, so we can omit writing "$* \mathsf{true}$" in these definitions.

The proof of $\{\mathbf{I}_{sl}(\mathcal{A})\}$ AV_SAL ; AV_SAL$[\xi]$ $\{\mathbf{I}_{sl}(=_{\{a_l\}})\}$ is not too difficult to work out (and can be found in Appendix B).

Close to Separation Logic is the *Relational Separation Logic* [46]. Relational separation logic is a logic to specify relations between two pointer programs and prove their specifications. We remark that is also possible to express TS-$(\mathcal{I}_1, \mathcal{I}_2)$-security using this logic. As relational separation logic deals simultaneously with two programs in tuples of the form $\{P\}\ \begin{matrix} S \\ S' \end{matrix}\ \{Q\}$, there is no need to use self-composition since the "quadruple" can hold separately the program an its renamed copy.

9 Temporal Logics

Computation Tree Logic (CTL for short) [10] 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 EF ϕ 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

$$\begin{array}{lll} c \models \mathsf{EF}\,\phi & \text{iff} & \exists \rho : \rho_0 = c : \exists i : \rho_i \models \phi \\ c \models \mathsf{AF}\,\phi & \text{iff} & \forall \rho : \rho_0 = c : \exists i : \rho_i \models \phi \end{array}$$

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 $c \models \psi$).

In this section we impose an extra requirement on the composition $S_1 \triangleright S_2$ that allows to syntactically identify the moment of the execution in which S_1 has just finished executing but S_2 has not yet started:

(c) $(S_1 \triangleright S_2, \mu) \rightsquigarrow^* (S_2, \mu')$ implies $(S_1, \mu) \rightsquigarrow^* (\surd, \mu')$.

Though this requirement is not strictly necessary, it is nonetheless convenient to keep simple the CTL formulas that characterize security.

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 $\mu \sim^{\mathcal{I}}_{\xi} \mu$. 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 characterize separable memories like we did in Section 8.)

In the following we give characterisations of non-interference in CTL.

Proposition 5. *A program S is TS $(\mathcal{I}_1, \mathcal{I}_2)$-secure if and only if $S \triangleright S[\xi]$ satisfies*

$$\mathsf{Ind}[\mathcal{I}_1] \rightarrow \mathsf{AG}(\mathsf{mid} \rightarrow \mathsf{EF}(\mathsf{end} \wedge \mathsf{Ind}[\mathcal{I}_2])). \tag{11}$$

Property (11) states that "whenever the initial state is indistinguishable, every time $S[\xi]$ is reached (and hence S terminates), there is an execution that leads to a terminating indistinguishable state". The CTL characterization of TS security given by Proposition 5 can be proven using Corollary 1 (see Appendix C).

Requirement (c) is necessary so that formula (11) is not confounded with the satisfaction of mid on several states along a single execution. For instance, this confusion appears in the case of program S^W, defined below, if it is self composed using only sequential composition.

$$\begin{array}{l} \mathsf{while}\ x < 2\ \ \mathsf{do} \\ \quad \mathsf{if}\ \ [\!]\ x = 0 \rightarrow\ \ x := 2 \\ \qquad\ [\!]\ \mathsf{true} \rightarrow\ \ x := 1\ \ \ \mathsf{fi} \\ \mathsf{od} \end{array}$$

Notice that, for instance, $(S^W; S^W[\xi], [x{\mapsto}0, x'{\mapsto}0]) \rightsquigarrow^* (S^W[\xi], [x{\mapsto}2, x'{\mapsto}1])$ with $(S^W[\xi], [x{\mapsto}2, x'{\mapsto}1]) \models \mathsf{mid}$, but configuration $(S^W[\xi], [x{\mapsto}2, x'{\mapsto}1])$ has already execute program S^W for a while. This contradicts the spirit of proposition mid. To avoid this situation the composition $S^W \triangleright S^W[\xi]$ may be defined using, for example, the skip instruction (or a trivial assignment) in between: $S^W; \mathsf{skip}; S^W[\xi]$. Then mid is defined to hold only in configurations of the form $(\mathsf{skip}; S^W[\xi], \mu)$ for some memory μ.

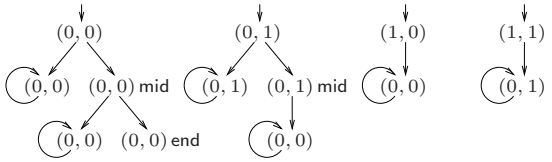
For the termination insensitive case, first notice that a program does not terminate if no execution reaches a terminating state. That is, $\neg\exists \mu' : (S, \mu) \rightsquigarrow^* (\surd, \mu')$, or equivalently $\forall S', \mu' : (S, \mu) \rightsquigarrow^* (S', \mu') : S' \neq \surd$. Therefore, program S does not terminate in μ if and only if $(S, \mu) \models \mathsf{AG}\,\neg\mathsf{end}$. The TI security characterization in CTL is obtained from (11) by allowing non-termination as follows.

Proposition 6. *A program S is TI $(\mathcal{I}_1, \mathcal{I}_2)$-secure if and only if $S \triangleright S[\xi]$ satisfies*

$$\mathsf{Ind}[\mathcal{I}_1] \rightarrow \mathsf{AG}(\mathsf{mid} \rightarrow ((\mathsf{AG}\,\neg\mathsf{end}) \vee \mathsf{EF}(\mathsf{end} \wedge \mathsf{Ind}[\mathcal{I}_2]))), \tag{12}$$

Property (12) says that "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 proof of Proposition 6 is similar to that of Proposition 5.

(a)



(b)

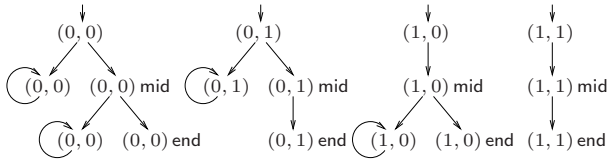


Figure 2: Automata for programs of Example 9

Example 9. Let y_h be a confidential variable in the following programs (borrowed from [29]):

(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 non-interferent [43, 29], that is, whether they are $(=_L, =_L)$-secure. We use CTL and for this we set $\mathsf{Ind}[=_\emptyset] \equiv \mathsf{true}$. The automaton 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 [29], program (a) *does* leak information: if it terminates, y_h must be equal to 0 at the beginning of the program. 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. Instead, the formula is valid in program (b).

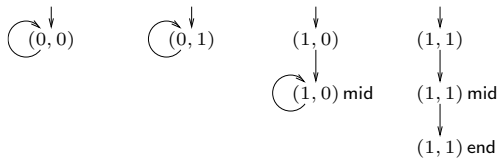


Figure 3: Automata for program of Example 10

Linear Temporal Logic characterization of security. A similar characterization can be given for Linear Temporal Logic (LTL) [31] but limited to deterministic programs. Like CTL, LTL extends propositional logic with modal operations. However these modalities refer only to properties of single executions disregarding path 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 characterized 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 characterized 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 in a non-deterministic setting cannot be characterized with LTL (at least using our technique).

Example 10. Let y_h be a confidential variable in program $\mathsf{while}\ y_h{=}0\ \mathsf{do}\ y_h := 0\ \mathsf{od}$. We check non-interference, that is, $(=_L, =_L)$-security. Then $\mathsf{Ind}(=_\emptyset) \equiv \mathsf{true}$ because there are no low variables. Figure 3 depicts the automaton for $\mathsf{while}\ y_h{=}0\ \mathsf{do}\ y_h := 0\ \mathsf{od}; \mathsf{while}\ y'_h{=}0\ \mathsf{do}\ y'_h := 0\ \mathsf{od}$ where variables take only value 0 or 1. Like before, a state is depicted with a tuple (d, d') containing the values of y_h and y'_h respectively, and the validity of mid and end is indicated next to the state. Notice that while the TI formula holds (in fact $\mathsf{true} \rightarrow \mathsf{G}(\mathsf{end} \rightarrow \mathsf{true}) \equiv \mathsf{true}$), the TS formula $\mathsf{true} \rightarrow ((\mathsf{F}\,\mathsf{mid}) \rightarrow (\mathsf{F}(\mathsf{end} \wedge \mathsf{true}))$, does not hold if $y_h{=}1$ and $y'_h{=}0$ (third automaton from the left).

Termination. Example 9 anticipates certain subtleties arising from termination. It has been argued that program (b) still leaks information [29]. 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 extent, our characterization of TS $(\mathcal{I}_1, \mathcal{I}_2)$-security fails.

So far, we have considered *strict* non-termination: $(S, \mu)\perp$ states that S *does not* terminate in μ. A notion of *possibilistic* non-termination can also be given: let $(S, \mu)\nearrow$ state that there is an execution of S from memory μ which 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 $\mu_1 \sim_{id}^{\mathcal{I}_1} \mu_2$,

$$(o) \quad \forall \mu_1' : (S, \mu_1) \rightsquigarrow^* (\surd, \mu_1') \Rightarrow (\exists \mu_2' : (S, \mu_2) \rightsquigarrow^* (\surd, \mu_2') \wedge \mu_1' \sim_{id}^{\mathcal{I}_2} \mu_2').$$

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

$$\begin{array}{ll} (i) \;\; (S, \mu_1)\perp \Rightarrow (S, \mu_2)\perp & (iii) \;\; (S, \mu_1)\nearrow \Rightarrow (S, \mu_2)\perp \\ (ii) \;\; (S, \mu_1)\perp \Rightarrow (S, \mu_2)\nearrow & (iv) \;\; (S, \mu_1)\nearrow \Rightarrow (S, \mu_2)\nearrow \end{array}$$

Since $\neg(S, \mu)\perp$ 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 $(S, \mu)\perp$ 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 9 is insecure under this condition. But so is

$$\mathsf{if}\ [\!]\ \mathsf{true} \rightarrow \mathsf{while\ true\ do}\ h := h\ \mathsf{od}\ \ [\!]\ \ \mathsf{true} \rightarrow h := h\ \mathsf{fi} \qquad (13)$$

which evidently does not reveal any information assuming an scheduler that makes nondeterministic choices without accessing high 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 9(b) as insecure, but considers (13) to be secure.

The following CTL formulas characterize 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.

10 Related work

Type-based analyzes are by far the most common form of enforcing information-flow policies of programs, see e.g. [40]. There is a however a growing body of work that pursues similar goals to ours, namely to enforce non-interference using logical methods.

Works using self-composition Our work is inspired from earlier results of Joshi and Leino [29], 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.

The idea of self-composition also appears in the work of Darvas, Hähnle and Sands [16], who suggest that dynamic logic can be used to verify non-interference policies (termination sensitive and termination insensitive, and modulo declassification) for imperative programs. Their work shares many motivations with ours, but they focus on a specific programming language and program logic; furthermore, they do not discuss completeness issues.

The idea of self-composition has been explored further in a series of recent works. For example, Terauchi and Aiken [44] have used this idea to formulate a notion of relaxed non-interference. They also propose a type-directed transformation as a solution for some safety analysis tools that try to solve problems semantically, and whose analysis will eventually not terminate in presence of certain predicates, e.g. predicates including complex arithmetic. In a nutshell, the type-directed transformation of programs does not self-compose branching statements depending on public variables, and makes a kind of copy propagation optimization to self-composed assignments with low expressions to variables. For example, if the following program has

public variables x and z and confidential variable y,

$$\mathsf{if}\ (x > z)\ \mathsf{then}\ x := z\ \mathsf{else}\ x := y\ \mathsf{fi}$$

the program is transformed into

$$\mathsf{if}\ (x > z)\ \mathsf{then}\ x := z\ \ ; x' := z'\ \mathsf{else}\ x := y\ \ ; x' := y'\ \mathsf{fi}$$

In addition, Terauchi and Aiken introduce the class of 2-safety properties, which can be reduced to safety properties by composing the program with itself, and show that non-interference is an instance of a 2-safety property. More recently, Clarkson and Schneider [11] have generalized this work to consider hyperproperties, that cover both liveness and safety, and that generalize 2-safety to n-safety.

Moving towards realistic programming languages, Jacobs and Warnier [28] 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. More recently, Dufay et al. [19] have experimented with verification of information flow for Java programs using self-composition and JML specifications [30]; more precisely, they have used the Krakatoa tool [32] to validate data mining algorithms. Their work is more oriented towards applications and does not justify formally self-composition. However, Naumann [34] has recently systematized and formally justified the modeling of information flow policies for Java programs using JML specifications. The work of Naumann is heavily inspired from earlier work by Benton [9], who develops a relational Hoare logic for a simple imperative language.

In a concurrent setting, Huisman *et al* [25] have recently proposed a characterization of observational determinism [49] using self-composition. Their characterization uses temporal logics and is thus amenable to model-checking after a suitable program abstraction has been constructed. On a negative side, Alur et al [1] establish that a more general notion of confidentiality than non-interference cannot be characterized using self-composition.

More recently, a number of works have explored self-composition in connection with quantitative analysis of information flow. For example, Backes, Kopf, and Rybalchenko [6] use ideas of self-composition to automatically discover paths that yield information leaks, and use this information to quantify the amount of information leakage. More recently, Yasuoka and Terauchi [47] have explored the possibility of expressing quantitative information flow policies as k-safety properties.

Dedicated logics and decision procedures Andrews and Reitman [4] were among the first to develop proof rules to reason about information flow for a concurrent imperative language. More recently, there have been several works that use specific logics for enforcing non-interference.

Using the framework of abstract interpretation, Giaccobazzi and Mastroeni [21] provide a proof method to prove abstract non-interference. This line of work has been extended more recently to Java bytecode by Zanardini [48].

Using a dedicated logic based on the notion of independence, Amtoft *et al* [2] propose a logic for information flow analysis for object-oriented programs. Their logic deals with pointer analysis using region analysis and employs independence assertions to describe non-interference. This approach has been recently extended to declassification [7], and to conditional information flow [3].

Dam [15] provides a sound and complete proof procedure to verify a notion of non-interference based on strong bisimulation for the While language with parallel composition of Section 2. In addition, he shows the decidability of non-interference under the assumption that the set of values is finite.

Certifying compilation for information flow Motivated by the possibility to automate parts of proofs of non-interference based on self-composition, our conference paper briefly discussed the relationship between type systems and program logics, and established the validity of hybrid rules that could be used to embed type derivations into logic derivations. For example, 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 [45] and from our characterization of non-interference that the following rule is valid:

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

More recently, several authors have explored the interplay between type systems and program logics further, and provided a systematic method to derive logical proofs of non-interference from type derivations. In particular, Beringer and Hofmann have explored a semantical notion of self-composition, that dispenses from reasoning on a self-composed program, and showed how to generate automatically formal proofs of non-interference

from valid typing derivations in several information flow type systems, including flow-sensitive type systems and type systems for fragments of Java. In a similar spirit, Hähnle *et al* [23] encode the flow sensitive type system of Hunt and Sands [26] into an extension of dynamic logic with updates.

11 Conclusion

We have developed a general theory of self-composition to prove that programs are non-interfering. Being based on logic, self-composition is expressive and does not require to prove the soundness of type systems. One natural direction for further research is to provide similar characterizations for other notions of non-interference, and perhaps for other security properties such as anonymity.

Acknowledgments We thank an anonymous referee for his/her thorough work on reviewing our submission which let us significantly improve the quality of our paper.

References

[1] R. Alur, P. Cerný, and S. Zdancewic. Preserving secrecy under refinement. In *33rd International Colloquium on Automata, Languages and Programming (ICALP)*, volume 4052 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 2006.

[2] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *Proceedings of POPL'06*, pages 91–102. ACM Press, 2006.

[3] T. Amtoft and A. Banerjee. Verification condition generation for conditional information flow. In *5th ACM Workshop on Formal Methods in Security Engineering (FMSE'07), George Mason University*, pages 2–11. ACM, November 2007. The full paper appears as Technical report 2007-2, Department of Computing and Information Sciences, Kansas State University, August 2007.

[4] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–75, January 1980.

[5] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. *ACM Trans. Program. Lang. Syst.*, 2(1):56–76, 1980.

[6] M. Backes, B. Köpf, and A. Rybalchenko. Automatic Discovery and Quantification of Information Leaks. In *Proc. 30th IEEE Symposium on Security and Privacy (S& P '09)*, pages 141–153. IEEE, 2009.

[7] A. Banerjee, D. Naumann, and S. Rosenberg. Towards a logical account of declassification. In *Proceedings of PLAS'07*, pages 61–65. ACM Press, 2007.

[8] G. Barthe, P.R. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Proceedings of CSFW'04*, pages 100–114. IEEE Press, 2004.

[9] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of POPL'04*, pages 14–25. ACM Press, 2004.

[10] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[11] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *CSF*, pages 51–65, 2008.

[12] E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.

[13] E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[14] S.A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.

[15] M. Dam. Decidability and proof systems for language-based noninterference relations. In *Proceedings of POPL'06*, pages 67–78. ACM Press, 2006.

[16] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Security in Pervasive Computing*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer, 2005. Preliminary version in the informal proceedings of WITS'03.

[17] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[18] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[19] G. Dufay, A. P. Felty, and S. Matwin. Privacy-sensitive information flow with JML. In R. Nieuwenhuis, editor, *Proceedings of CADE'05*, volume 3632 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2005.

[20] 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.

[21] R. Giacobazzi and I. Mastroeni. Proving abstract non-interference. In *Annual Conference of the European Association for Computer Science Logic (CSL'04).*, volume 3210 of *Lecture Notes in Computer Science*, pages 280–294. Springer, 2004.

[22] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of SOSP'82*, pages 11–22. IEEE Press, 1982.

[23] R. Hähnle, J. Pan, P. Rümmer, and D. Walter. Integration of a security type system into a program logic. In *Proc. 2nd Symposium on Trustworthy Global Computing*, volume 4661 of *Lecture Notes in Computer Science*, pages 116–131. Springer, 2007.

[24] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969.

[25] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Proceedings of CSFW'06*, pages 3–15. IEEE Computer Society Press, 2006.

[26] S. Hunt and D. Sands. On flow-sensitive security types. *ACM SIGPLAN Notices–Proceedings of POPL 2006*, 41(1):79–90, January 2006.

[27] 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.

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

[29] R. Joshi and K.R.M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1-3):113–138, 2000.

[30] G.T. Leavens, A.L. 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.

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

[32] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated with JML annotations. *Journal of Logic and Algebraic Programming*, 58:89–106, 2004.

[33] John Mclean. A general theory of composition for trace sets closed under selective interleaving functions. In *In Proc. IEEE Symposium on Security and Privacy*, pages 79–93, 1994.

[34] D. Naumann. From coupling relations to mated invariants for checking information flow (extended abstract). In D. Gollmann and A. Sabelfeld, editors, *Proceedings of ESORICS'06*, volume 4189 of *Lecture Notes in Computer Science*, pages 279–296. Springer, 2006.

[35] G.C. Necula. Proof-carrying code. In *Proc. of the* 25^{th} *ACM Symp. on Principles of Programming Languages,* Paris, pages 106–119, 1997.

[36] G.C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.

[37] François Pottier. A simple view of type-secure information flow in the pi-calculus. In *CSFW*, pages 320–330. IEEE Computer Society, 2002.

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

[39] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, 2002.

[40] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21:5–19, January 2003.

[41] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, Lecture Notes in Computer Science. Springer, 2004.

[42] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *CSFW*, pages 255–269, 2005.

[43] G. Smith and D. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *Proc. of the* 25^{th} *ACM Symp. on Principles of Programming Languages*, pages 355–364, San Diego, California, January 1998.

[44] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In C. Hankin and I. Siveroni, editors, *Static Analysis Symposium*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2005.

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

[46] H. Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007.

[47] Hirotoshi Yasuoka and Tachio Terauchi. Quantitative information flow - verification hardness and possibilities. In *Proc. CSF'10*. IEEE, 2010.

[48] D. Zanardini. Abstract Non-Interference in a fragment of Java bytecode. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 1822–1826, Dijon, France, April 2006. ACM Press, New York.

[49] S. Zdancewic and A.C. Myers. Observational determinism for concurrent program security. In *Proceedings of CSFW'03*, pages 29–43, Pacific Grove, California, USA, June 2003. IEEE Press.

A Appendix

A Proof of Theorem 3

A.1 Termination Sensitive Case

We first calculate:

S is TS $(\mathcal{I}_1, \mathcal{I}_2)$-secure
iff {Cor. 1}
$S \overset{\xi,\mathcal{I}_1}{\underset{\xi,\mathcal{I}_2}{\rhd\!\approx}} S[\xi]$
iff {Def. 2}
$\forall \mu_1, \mu_2, \mu_1' : \mathsf{var}(\mu_1) = \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$\quad \mu_1 \oplus \mu_2 \sim_\xi^{\mathcal{I}_1} \mu_1 \oplus \mu_2 \ \wedge \ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (S[\xi], \mu_1' \oplus \mu_2)$
$\quad \Rightarrow \exists \mu_2' : \mathsf{var}(\mu_2') = \mathsf{var}(S)' :$
$\qquad (S[\xi], \mu_1' \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'$
iff {Prop. (a) and (b) of $\triangleright$ and Fact 1.2 for implication $\Leftarrow$}
$\forall \mu_1, \mu_2, \mu_1' : \mathsf{var}(\mu_1) = \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$\quad \mu_1 \oplus \mu_2 \sim_\xi^{\mathcal{I}_1} \mu_1 \oplus \mu_2 \ \wedge \ (S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2)$
$\quad \Rightarrow \exists \mu_2' : \mathsf{var}(\mu_2') = \mathsf{var}(S)' :$
$\qquad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'$
iff {Logic}
$\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$\quad \mu_1 \oplus \mu_2 \sim_\xi^{\mathcal{I}_1} \mu_1 \oplus \mu_2$
$\quad\ \Rightarrow \forall \mu_1' : \mathsf{var}(\mu_1') = \mathsf{var}(S) :$
$\qquad \neg(\ (S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2)\ ) \vee$
$\qquad \exists \mu_2' : \mathsf{var}(\mu_2') = \mathsf{var}(S)' :$
$\qquad\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'$ (14)

Starting in the other direction of the implication, we calculate

$$
\begin{array}{l}
\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' : \\
\quad \mu_1 \oplus \mu_2 \sim_\xi^{\mathcal{I}_1} \mu_1 \oplus \mu_2 \ \wedge \ \exists \mu_1'' : (S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1'' \oplus \mu_2) \\
\quad\quad \Rightarrow \exists \mu_1', \mu_2' : \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2') = \mathsf{var}(S)' : \\
\quad\quad\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2' \\
\textbf{iff} \quad \{\text{Logic}\} \\
\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' : \\
\quad \mu_1 \oplus \mu_2 \sim_\xi^{\mathcal{I}_1} \mu_1 \oplus \mu_2 \\
\quad\quad \Rightarrow (S, \mu_1 \oplus \mu_2)\bot \vee \\
\quad\quad\quad \exists \mu_1', \mu_2' : \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2') = \mathsf{var}(S)' : \\
\quad\quad\quad\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'
\end{array}
\Bigg\} (15)
$$

We now show that (14) and (15) are equivalent considering two different cases. For the first case suppose $(S, \mu_1 \oplus \mu_2)\bot$. It is easy to check that both (14) and (15) hold under this hypothesis.

Now suppose $\neg((S, \mu_1 \oplus \mu_2)\bot)$. Because S is deterministic, there must exist a unique memory μ such that $(S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu)$. Moreover, because of Fact 1.1, there is a unique μ_1'' with $\mathsf{var}(\mu_1'') = \mathsf{var}(S)$ such that $(S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1'' \oplus \mu_2)$. Under this hypothesis, we then calculate:

$$
\begin{array}{l}
(15) \\
\textbf{iff} \quad \{\neg((S, \mu_1 \oplus \mu_2)\bot), \text{Prop. (a) and (b) of } \triangleright \text{ and uniqueness of } \mu_1''\} \\
(S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1'' \oplus \mu_2) \\
\wedge \ \exists \mu_2' : \mathsf{var}(\mu_2') = \mathsf{var}(S)' : (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1'' \oplus \mu_2') \ \wedge \ \mu_1'' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1'' \oplus \mu_2' \\
\textbf{iff} \quad \{\text{First conjunct holds because uniqueness of } \mu_1''\} \\
\forall \mu_1' : \mathsf{var}(\mu_1') = \mathsf{var}(S) \wedge \mu_1' \neq \mu_1'' : \neg(\ (S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2)\ ) \\
\wedge \ (S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1'' \oplus \mu_2) \\
\wedge \ \exists \mu_2' : \mathsf{var}(\mu_2') = \mathsf{var}(S)' : (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1'' \oplus \mu_2') \ \wedge \ \mu_1'' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1'' \oplus \mu_2' \\
\textbf{iff} \quad \{\text{Logic and determinism for implication} \Leftarrow\} \\
(14)
\end{array}
$$

We can adapt this proof to work as well for the termination sensitive case of Theorem 4. First observe that if S is an observationally deterministic program, then either $(S, \mu_1 \oplus \mu_2)\bot$ or for all $\mu_1', \mu_1'' \in \Theta$, $\mathsf{v}(\mu_1', x) = \mathsf{v}(\mu_1'', x)$ for every $x \in \mathsf{var}(S)$, where $\Theta = \{\mu \mid (S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu \oplus \mu_2)\}$. Moreover, notice that for all $\mu_1', \mu_1'' \in \Theta$,

$$
\mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2' \quad \text{iff} \quad \mu_1'' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1'' \oplus \mu_2' \tag{16}
$$

Then, the only difference with the previous proof happens in the last case, in which $\neg((S, \mu_1 \oplus \mu_2)\bot)$. For this case we proceed as follows.

(15)
iff $\{\neg((S, \mu_1 \oplus \mu_2)\bot)\}$
$\exists \mu_1', \mu_2' : \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2') = \mathsf{var}(S)' :$
$\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'$
iff {Prop. (b) of $\triangleright$ and Fact 1.2}
$\exists \mu_1', \mu_2' : \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2') = \mathsf{var}(S)' :$
$\quad (S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2) \wedge$
$\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'$
iff $\{\Theta \neq \emptyset,$ observation (16) and logic$\}$
$\forall \mu_1' : \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge \ \mu_1' \in \Theta :$
$\quad (S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2) \wedge$
$\quad \exists \mu_2' : \mathsf{var}(\mu_2') = \mathsf{var}(S)' : (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'$
iff {First conjunct holds by definition of Θ}
$\forall \mu_1' : \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge \ \mu_1' \notin \Theta : \neg(\ (S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2)\ ) \wedge$
$\forall \mu_1' : \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge \ \mu_1' \in \Theta : (S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2) \wedge$
$\quad \exists \mu_2' : \mathsf{var}(\mu_2') = \mathsf{var}(S)' : (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'$
iff {Logic and definition of Θ for implication $\Leftarrow$}
(14)

A.2 Termination Insensitive Case

S is TI $(\mathcal{I}_1, \mathcal{I}_2)$-secure
iff {Corollary 1}
$S \overset{\mathcal{I}_1}{\underset{\mathcal{I}_2}{\rhd\sim}} S[\xi]$
iff {Def. 2}
$\forall \mu_1, \mu_2, \mu_1' : \mathsf{var}(\mu_1) = \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$\quad \mu_1 \oplus \mu_2 \sim_\xi^{\mathcal{I}_1} \mu_1 \oplus \mu_2 \ \wedge \ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (S[\xi], \mu_1' \oplus \mu_2)$
$\quad \Rightarrow \exists \mu_2' : \mathsf{var}(\mu_2') = \mathsf{var}(S)' :$
$\quad\quad \big((S[\xi], \mu_1' \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'\big)$
$\quad\quad \vee \ (S[\xi], \mu_1' \oplus \mu_2)\bot$
iff {Prop. (a) and (b) of $\triangleright$ and Fact 1.2 for implication $\Leftarrow$}
$\forall \mu_1, \mu_2, \mu_1' : \mathsf{var}(\mu_1) = \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$\quad \mu_1 \oplus \mu_2 \sim_\xi^{\mathcal{I}_1} \mu_1 \oplus \mu_2 \ \wedge \ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (S[\xi], \mu_1' \oplus \mu_2)$
$\quad \Rightarrow \exists \mu_2' : \mathsf{var}(\mu_2') = \mathsf{var}(S)' :$
$\quad\quad \big((S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'\big)$
$\quad\quad \vee \ (S[\xi], \mu_1' \oplus \mu_2)\bot$
iff {Claim 1 below}
$\forall \mu_1, \mu_2, \mu_1' : \mathsf{var}(\mu_1) = \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$\quad \mu_1 \oplus \mu_2 \sim_\xi^{\mathcal{I}_1} \mu_1 \oplus \mu_2 \ \wedge \ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (S[\xi], \mu_1' \oplus \mu_2)$
$\quad \Rightarrow \exists \mu_2' : \mathsf{var}(\mu_2') = \mathsf{var}(S)' :$
$\quad\quad \big((S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'\big)$
$\quad\quad \vee \ \forall \mu_2'' : \mathsf{var}(\mu_2'') = \mathsf{var}(S)' : \neg(\ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2'')\ )$
iff {Logic}
$\forall \mu_1, \mu_2, \mu_1', \mu_2'' : \mathsf{var}(\mu_1) = \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(\mu_2'') = \mathsf{var}(S)' :$
$\quad \big(\mu_1 \oplus \mu_2 \sim_\xi^{\mathcal{I}_1} \mu_1 \oplus \mu_2 \ \wedge \ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (S[\xi], \mu_1' \oplus \mu_2)$
$\quad\quad\quad\quad\quad\quad \wedge \ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2'')\big)$
$\quad \Rightarrow \exists \mu_2' : \mathsf{var}(\mu_2') = \mathsf{var}(S)' :$
$\quad\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'$
iff {Prop. (a) and (b) of $\triangleright$ and Fact 1.2}
$\forall \mu_1, \mu_2, \mu_1', \mu_2'' : \mathsf{var}(\mu_1) = \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(\mu_2'') = \mathsf{var}(S)' :$
$\quad \big(\mu_1 \oplus \mu_2 \sim_\xi^{\mathcal{I}_1} \mu_1 \oplus \mu_2 \ \wedge \ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2'')\big)$
$\quad \Rightarrow \exists \mu_2' : \mathsf{var}(\mu_2') = \mathsf{var}(S)' :$
$\quad\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'$
iff {Logic}
$\forall \mu_1, \mu_2, \mu_1', \mu_2'' : \mathsf{var}(\mu_1) = \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(\mu_2'') = \mathsf{var}(S)' :$
$\quad \big(\mu_1 \oplus \mu_2 \sim_\xi^{\mathcal{I}_1} \mu_1 \oplus \mu_2 \ \wedge \ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2'')\big)$
$\quad \Rightarrow \exists \mu_2' : \mathsf{var}(\mu_2') = \mathsf{var}(S)' :$
$\quad\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2'') \ \wedge$
$\quad\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'$
iff {Determinism ($\mu_2' = \mu_2''$) and logic} (17)
$\forall \mu_1, \mu_2, \mu_1', \mu_2' : \mathsf{var}(\mu_1) = \mathsf{var}(\mu_1') = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(\mu_2') = \mathsf{var}(S)' :$
$\quad \big(\mu_1 \oplus \mu_2 \sim_\xi^{\mathcal{I}_1} \mu_1 \oplus \mu_2 \ \wedge \ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2')\big)$
$\quad \Rightarrow \mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'$

Step (17) can equally be justified by observational equivalence, in this case taking into account that $\mathsf{v}(\mu_2', x) = \mathsf{v}(\mu_2'', x)$ for every $x \in \mathsf{var}(S)'$, and hence $\mu_1' \oplus \mu_2' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2'$ iff $\mu_1' \oplus \mu_2'' \sim_\xi^{\mathcal{I}_2} \mu_1' \oplus \mu_2''$. This proves the termination insensitive case of Theorem 4.

Claim 1. *Let* μ_1, μ_2, μ_1' *such that* $\mathsf{var}(\mu_1) = \mathsf{var}(\mu_1') = \mathsf{var}(S)$ *and* $\mathsf{var}(\mu_2) = \mathsf{var}(S)'$. *If* $(S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (S[\xi], \mu_1' \oplus \mu_2)$, *then* $(S[\xi], \mu_1' \oplus \mu_2)\bot$ *iff* $\forall \mu_2'' : \mathsf{var}(\mu_2'') = \mathsf{var}(S)' : \neg(\ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2'')\ )$.

Proof. First notice that $(S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2)$ because of Prop. (b) of $\triangleright$ and Fact 1.2. Then we have:

$(S[\xi], \mu_1' \oplus \mu_2)\bot$
iff {Fact 1.1}
$\neg\exists \mu_2'' : \mathsf{var}(\mu_2'') = \mathsf{var}(S)' : (S[\xi], \mu_1' \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2'')$
iff {By previous observation after hypothesis of the claim and logic}
$(S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2)$
$\Rightarrow \forall \mu_2'' : \mathsf{var}(\mu_2'') = \mathsf{var}(S)' : \neg(\ (S[\xi], \mu_1' \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2'')\ )$
iff {Logic}
$\forall \mu_2'' : \mathsf{var}(\mu_2'') = \mathsf{var}(S)' :$
$\quad \neg(\ (S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2)\ \wedge\ (S[\xi], \mu_1' \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2'')\ )$
iff {Prop. (b) of $\triangleright$ and Fact 1.2 for implication $\Leftarrow$}
$\forall \mu_2'' : \mathsf{var}(\mu_2'') = \mathsf{var}(S)' : \neg(\ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2'')\ )$

B Proof for Example 8

The invariant for the while do in AV_SAL is

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

where $+\!\!+$ denotes concatenation.

The intuition behind the invariant is as follows. First, the general indistinguishability invariant has to hold (last line in the equation). Then, it splits the memory in two parts and it basically focus on the "non-primed" part of the memory (the one confined to AV_SAL). This part states that the original salary list (represented here by list ps) can be split in two salary lists, ps_{prev} and ps_{mis}. ps_{prev} contains the elements that have already been accounted while ps_{mis} contains those still to be accounted. This (partial) accounting of the salaries involve two operations: a summation, which is stored in variable s, and an element counting, which is stored in variable n.

In this way, at the end of the loop, p is nil implying that $ps_{mis} = [\,]$ and hence $ps_{prev} = ps$. Therefore s will be equal the sum of all the salaries and n the length of the original list, i.e. the quantities of summed salaries.

For the verification of $\{\mathbf{I}_{sl}(\mathcal{A})\}$ AV_SAL ; AV_SAL$[\xi]$ $\{\mathbf{I}_{sl}(=_{\{a_l\}})\}$ we focus on the second part of the algorithm. The first part basically repeats the same proof, which we left for the interested reader. We only omit some few proof obligations.

$$\left\{ \begin{array}{l} \exists ps, ps' : \quad \mathsf{list}.ps.lsalaries * \mathsf{list}.ps'.lsalaries' \\ \qquad\qquad\quad \wedge \ \frac{\sum ps}{|ps|} = \frac{\sum ps'}{|ps'|} \end{array} \right\}$$

// *AV_SAL: the program itself as a first part*
// *of the composed program*

$p := lsalaries$
$s := 0$
$n := 0$

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

```
while p ≠ nil  do
   n := n + 1
   s_aux := p.salary
   s := s + s_aux
   p_aux := p.next
   p := p_aux
od
```

$a_l := s/n$

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

// *AV_SAL*$[\xi]$*: the renamed part of the program*

$p' := lsalaries'$

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

$s' := 0$

$$\left\{ \begin{array}{l} \exists ps, ps' : \mathsf{list}.ps.lsalaries \\ \qquad\qquad * \ (\ \mathsf{list}.ps'.lsalaries' \wedge \mathsf{list}.ps'.p' \wedge s' = 0 \ ) \\ \qquad\qquad \wedge \ \big(a_l = \frac{\sum ps'}{|ps'|}\big) \end{array} \right\}$$

$n' := 0$

$$\left\{ \begin{array}{l} \exists ps, ps' : \mathsf{list}.ps.lsalaries \\ \qquad\qquad * \ (\ \mathsf{list}.ps'.lsalaries' \wedge \mathsf{list}.ps'.p' \wedge s' = 0 \wedge n' = 0 \ ) \\ \qquad\qquad \wedge \ \big(a_l = \frac{\sum ps'}{|ps'|}\big) \end{array} \right\}$$

// *take* $ps'_{prev} = [\,]$ *and* $ps'_{mis} = ps'$*, we obtain the invariant*

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

while $p' \neq \mathsf{nil}$ do

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

// $p' \neq nil$ *allows to change* ps'_{mis} *by* $[a] +\!\!+ ps'_{mis}$

$$\left\{\begin{array}{l}\exists ps, ps' :\ \mathsf{list}.ps.lsalaries \\ \qquad * \ (\ \mathsf{list}.ps'.lsalaries' \\ \qquad\quad \wedge\ \exists ps'_{mis}, ps'_{prev}, a, q : (ps' = ps'_{prev} +\!\!+ [a] +\!\!+ ps'_{mis}) \\ \qquad\qquad\qquad\qquad \wedge\ p' \mapsto (a, q) \wedge \mathsf{list}.ps'_{mis}.q \\ \qquad\qquad\qquad\qquad \wedge\ (s' = \sum ps'_{prev}) \wedge (n' = |ps'_{prev}|) \\ \qquad\ ) \\ \qquad \wedge\ \left(a_l = \frac{\sum ps'}{|ps'|}\right)\end{array}\right\}$$

$n' := n' + 1$

$$\left\{\begin{array}{l}\exists ps, ps' :\ \mathsf{list}.ps.lsalaries \\ \qquad * \ (\ \mathsf{list}.ps'.lsalaries' \\ \qquad\quad \wedge\ \exists ps'_{mis}, ps'_{prev}, a, q : (ps' = ps'_{prev} +\!\!+ [a] +\!\!+ ps'_{mis}) \\ \qquad\qquad\qquad\qquad \wedge\ p' \mapsto (a, q) \wedge \mathsf{list}.ps'_{mis}.q \\ \qquad\qquad\qquad\qquad \wedge\ (s' = \sum ps'_{prev}) \wedge (n' = |ps'_{prev}| + 1) \\ \qquad\ ) \\ \qquad \wedge\ \left(a_l = \frac{\sum ps'}{|ps'|}\right)\end{array}\right\}$$

$s'_{aux} := p'.salary$

$$\left\{\begin{array}{l}\exists ps, ps' :\ \mathsf{list}.ps.lsalaries \\ \qquad * \ (\ \mathsf{list}.ps'.lsalaries' \\ \qquad\quad \wedge\ \exists ps'_{mis}, ps'_{prev}, a, q : (ps' = ps'_{prev} +\!\!+ [a] +\!\!+ ps'_{mis}) \\ \qquad\qquad\qquad\qquad \wedge\ p' \mapsto (a, q) \wedge \mathsf{list}.ps'_{mis}.q \wedge s_{aux} = a \\ \qquad\qquad\qquad\qquad \wedge\ (s' = \sum ps'_{prev}) \wedge (n' = |ps'_{prev}| + 1) \\ \qquad\ ) \\ \qquad \wedge\ \left(a_l = \frac{\sum ps'}{|ps'|}\right)\end{array}\right\}$$

$s' := s' + s'_{aux}$

$$\left\{\begin{array}{l} \exists ps, ps' : \mathsf{list}.ps.lsalaries \\ \qquad * \ (\ \mathsf{list}.ps'.lsalaries' \\ \qquad\quad \wedge \exists ps'_{mis}, ps'_{prev}, a, q : (ps' = ps'_{prev} +\!\!+ [a] +\!\!+ ps'_{mis}) \\ \qquad\qquad\qquad\qquad\quad \wedge\ p' \mapsto (a, q) \wedge \mathsf{list}.ps'_{mis}.q \\ \qquad\qquad\qquad\qquad\quad \wedge\ (s' = \sum ps'_{prev} + a) \wedge (n' = |ps'_{prev}| + 1) \\ \qquad\quad) \\ \qquad \wedge\ \left(a_l = \frac{\sum ps'}{|ps'|}\right) \end{array}\right\}$$

$p'_{aux} := p'.next$

$$\left\{\begin{array}{l} \exists ps, ps' : \mathsf{list}.ps.lsalaries \\ \qquad * \ (\ \mathsf{list}.ps'.lsalaries' \\ \qquad\quad \wedge \exists ps'_{mis}, ps'_{prev}, a, q : (ps' = ps'_{prev} +\!\!+ [a] +\!\!+ ps'_{mis}) \\ \qquad\qquad\qquad\qquad\quad \wedge\ p' \mapsto (a, q) \wedge \mathsf{list}.ps'_{mis}.q \wedge q = p'_{aux} \\ \qquad\qquad\qquad\qquad\quad \wedge\ (s' = \sum ps'_{prev} + a) \wedge (n' = |ps'_{prev}| + 1) \\ \qquad\quad) \\ \qquad \wedge\ \left(a_l = \frac{\sum ps'}{|ps'|}\right) \end{array}\right\}$$

// *take* $ps'_{prev} = ps'_{prev} +\!\!+ [a]$

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

$p' := p'_{aux}$

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

od

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

// $p' = nil$ *implies* $ps'_{mis} = [\,]$ *and* $ps'_{prev} = ps'$

$$\left\{\begin{array}{l} \exists ps, ps' : \mathsf{list}.ps.lsalaries \\ \qquad * \ (\ \mathsf{list}.ps'.lsalaries' \wedge (s' = \sum ps') \wedge (n' = |ps'|)\ ) \\ \qquad \wedge\ \left(a_l = \frac{\sum ps'}{|ps'|}\right) \end{array}\right\}$$

$$\left\{\exists ps, ps' : \mathsf{list}.ps.lsalaries * \mathsf{list}.ps'.lsalaries' \wedge a_l = \frac{s'}{n'}\right\}$$

$a'_l := s'/n'$
$\{\exists ps, ps' : \mathsf{list}.ps.lsalaries * \mathsf{list}.ps'.lsalaries' \wedge a_l = a'_l\}$

C Proofs for characterization of security in CTL – TS Case

S is TS $(\mathcal{I}_1, \mathcal{I}_2)$-secure
iff {Cor. 1}
$S \overset{\triangleright}{\approx}^{id,\mathcal{I}_1}_{id,\mathcal{I}_2} S[\xi]$
iff {Def. 2}
$\forall \mu_1, \mu_2, \mu'_1 : \mathsf{var}(\mu_1) = \mathsf{var}(\mu'_1) = \mathsf{var}(S) \wedge \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$\quad \mu_1 \oplus \mu_2 \sim^{\mathcal{I}_1}_{\xi} \mu_1 \oplus \mu_2 \ \wedge \ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (S[\xi], \mu'_1 \oplus \mu_2)$
$\quad \Rightarrow \exists \mu'_2 : \mathsf{var}(\mu'_2) = \mathsf{var}(S)' :$
$\qquad (S[\xi], \mu'_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu'_1 \oplus \mu'_2) \ \wedge \ \mu'_1 \oplus \mu'_2 \sim^{\mathcal{I}_2}_{\xi} \mu'_1 \oplus \mu'_2$
iff {Satisfaction of $\mathsf{Ind}, \mathsf{end}$, and Fact 1. 1}
$\forall \mu_1, \mu_2, \mu'_1 : \mathsf{var}(\mu_1) = \mathsf{var}(\mu'_1) = \mathsf{var}(S) \wedge \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$\quad \big(\ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \models \mathsf{Ind}[\mathcal{I}_1] \ \wedge \ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (S[\xi], \mu'_1 \oplus \mu_2) \ \big)$
$\quad \Rightarrow \exists c : (S[\xi], \mu'_1 \oplus \mu_2) \rightsquigarrow^* c \ \wedge \ c \models \mathsf{Ind}[\mathcal{I}_2] \wedge \mathsf{end}$
iff {Semantics of EF and logic}
$\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \models \mathsf{Ind}[\mathcal{I}_1]$
$\quad \Rightarrow \forall \mu'_1 : \mathsf{var}(\mu'_1) = \mathsf{var}(S) :$
$\qquad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (S[\xi], \mu'_1 \oplus \mu_2) \Rightarrow (S[\xi], \mu'_1 \oplus \mu_2) \models \mathsf{EF}(\mathsf{Ind}[\mathcal{I}_2] \wedge \mathsf{end})$
iff {Logic}
$\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \models \mathsf{Ind}[\mathcal{I}_1]$
$\quad \Rightarrow \forall \mu : (\ \exists \mu'_1 : \mathsf{var}(\mu'_1) = \mathsf{var}(S) : \mu = \mu'_1 \oplus \mu_2 \wedge (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (S[\xi], \mu'_1 \oplus \mu_2) \ )$
$\qquad \Rightarrow (S[\xi], \mu) \models \mathsf{EF}(\mathsf{Ind}[\mathcal{I}_2] \wedge \mathsf{end})$
iff {Prop. (a) of $\triangleright$}
$\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \models \mathsf{Ind}[\mathcal{I}_1]$
$\quad \Rightarrow \forall \mu : (\ \exists \mu'_1 : \mathsf{var}(\mu'_1) = \mathsf{var}(S) : \mu = \mu'_1 \oplus \mu_2 \wedge (S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu'_1 \oplus \mu_2) \ )$
$\qquad \Rightarrow (S[\xi], \mu) \models \mathsf{EF}(\mathsf{Ind}[\mathcal{I}_2] \wedge \mathsf{end})$
iff {Logic and Fact 1.1}
$\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \models \mathsf{Ind}[\mathcal{I}_1]$
$\quad \Rightarrow \forall \mu : (S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu) \ \Rightarrow \ (S[\xi], \mu) \models \mathsf{EF}(\mathsf{Ind}[\mathcal{I}_2] \wedge \mathsf{end})$
iff {Prop. (c) of $\triangleright$ (for $\Rightarrow$) and Prop. (a) of $\triangleright$ (for $\Leftarrow$)}
$\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \models \mathsf{Ind}[\mathcal{I}_1]$
$\quad \Rightarrow \forall \mu : (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (S[\xi], \mu) \ \Rightarrow \ (S[\xi], \mu) \models \mathsf{EF}(\mathsf{Ind}[\mathcal{I}_2] \wedge \mathsf{end})$
iff {Logic and Satisfaction of mid}
$\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \models \mathsf{Ind}[\mathcal{I}_1]$
$\quad \Rightarrow \forall c : \big(\ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* c \ \wedge \ c \models \mathsf{mid} \ \big) \ \Rightarrow \ c \models \mathsf{EF}(\mathsf{Ind}[\mathcal{I}_2] \wedge \mathsf{end})$

iff {Semantics of $\rightarrow$, and logic}
$\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \models \mathsf{Ind}[\mathcal{I}_1]$
$\quad \Rightarrow \forall c : (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* c \Rightarrow c \models (\ \mathsf{mid} \rightarrow \mathsf{EF}(\mathsf{Ind}[\mathcal{I}_2] \wedge \mathsf{end})\ )$
iff {Semantics of AG and $\rightarrow$}
$\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' :$
$\quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \ \models \ \mathsf{Ind}[\mathcal{I}_1] \rightarrow \mathsf{AG}(\ \mathsf{mid} \rightarrow \mathsf{EF}(\mathsf{Ind}[\mathcal{I}_2] \wedge \mathsf{end})\ )$
iff $\left\{ \begin{array}{l} S \models \Phi \text{ iff } \forall \mu : (S, \mu) \models \Phi \text{ by definition and, since here memory are func-} \\ \text{tions, } \forall \mu : \exists \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \wedge \mathsf{var}(\mu_2) = \mathsf{var}(S)' \wedge \mu = \mu_1 \oplus \mu_2 \end{array} \right\}$
$S \triangleright S[\xi] \ \models \ \mathsf{Ind}[\mathcal{I}_1] \rightarrow \mathsf{AG}(\ \mathsf{mid} \rightarrow \mathsf{EF}(\mathsf{Ind}[\mathcal{I}_2] \wedge \mathsf{end})\ )$

Contents