Math. Struct. in Comp. Science (2011), *vol.* 21, *pp.* 1207–1252. © Cambridge University Press 2011
doi:10.1017/S0960129511000193

Secure information flow by self-composition[†]

GILLES BARTHE[‡], PEDRO R. D'ARGENIO[§] and TAMARA REZK[¶]

[‡]*IMDEA Software Institute, Madrid, Spain*
Email: gilles.barthe@imdea.org
[§]*FaMAF, Universidad Nacional de Córdoba – CONICET, Córdoba, Argentina*
Email: dargenio@famaf.unc.edu.ar
[¶]*INRIA Sophia-Antipolis, INDES Project, France*
Email: tamara.rezk@inria.fr

Received 2 June 2008

Information flow policies are confidentiality policies that control information leakage through program execution. A common way 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 proved sound for each new variation of security policy and programming language for which secure information flow verification is desired.

In contrast, program logics offer a general mechanism for enforcing a variety of safety policies, and for this reason are favoured in Proof Carrying Code, which is a promising security architecture for mobile code. However, the encoding of information flow policies in program logics is not 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, which 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 policy verification, such as program logics and model checking, that are 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 including 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 the 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 authorised to unauthorised users. In order

[†] This work was partially supported by ReSeCo.
[‡] Gilles Barthe is partially supported by Spanish project TIN2009-14599 DESAFIOS 10, and Madrid Regional project 2009STIC-1465 PROMETIDOS, Pedro R. D'Argenio is partially supported by PICT 26135.
[§] Pedro R. D'Argenio is partially supported by PICT 26135.

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

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, which is a baseline information flow policy that guarantees the absence of information leakage. Informally, non-interference considers a partitioning of the program state into public and secret parts, and requires that no information is leaked from 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 – see below for a more formal definition. Non-interference and information flow policies have their roots in Cohen (1977; 1978), Denning and Denning (1977) and Goguen and Meseguer (1982), and recently they have attracted substantial interest within language-based security studies – see Sabelfeld and Myers (2003) for a survey. Other forms of information flow policies that are less strict and more appropriate than non-interference for use in practice include declassification policies – see Sabelfeld and Sands (2005) for a survey.

The current paper is concerned with the static enforcement of information flow policies in general. The currently prevailing method for enforcing such policies is through information flow type systems (Sabelfeld and Myers 2003). Clearly, type systems are attractive because they support automated, compositional verification. However, type systems are inherently not extensible: every modification of 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 other hand, logical verification methods are flexible and can be used to support several policies without the need to prove soundness repeatedly. It is precisely because of this ability of logic to support various policies that Proof Carrying Code (Necula 1997; Necula 1998) relies on logical verification to validate mobile code on the consumer side. Typically, the consumer infrastructure consists of a verification condition generator, which operates on programs annotated with safety annotations, and a certificate checker, which 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 analysers, which use type systems or static analyses to generate the safety annotations, and automatically generate a proof of the correctness of the program with respect to these annotations. In such cases, the certifying compiler relies crucially 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 proved in, for example, Mclean (1994)), 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 (Andrews and Reitman 1980; Denning and Denning 1977). However, their encoding requires us to extend the set of axioms of Hoare logic in order to account for

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

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 the security property needs to be adapted. More recently, Darvas *et al.* (2005) showed 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 authors 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 (2002) for non-interference in the pi-calculus, where the non-interference of two processes $P1$ and $P2$ is reduced to a property of 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. We are not clear whether this kind of technique extends easily to declassification policies.

In the conference version of the current paper (Barthe *et al.* 2004), we used the term 'self-composition' for 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 generalised in Terauchi and Aiken (2005) to the class of 2-safety properties and in Clarkson and Schneider (2008) 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 for specifying and verifying 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 the current paper is to build on self-composition to provide characterisations of information flow policies in programming and temporal logics. Our characterisations apply to many languages and different notions of security including some forms of declassification.

In order to provide some intuition, we will first consider a simple deterministic imperative language featuring sequential composition and equipped with an evaluation relation $\langle P, \mu\rangle \Downarrow v$, where P is a program and μ, v are memories, *viz.* maps from the program variables of P to values. Furthermore, we 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 $\vec{y}$ be the set of all its private variables. Termination-insensitive non-interference for P may be exressed by

$$[\langle P, \mu_1\rangle \Downarrow v_1 \ \wedge \ \langle P, \mu_2\rangle \Downarrow v_2 \ \wedge \ \mu_1 =_L \mu_2] \ \Rightarrow v_1 =_L v_2$$

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

for all memories μ, μ', v, v', where $=_L$ is the point-wise extension of equality on values to the 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 the same 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 $\langle P, \mu\rangle \Downarrow v \ \wedge \ \langle P', \mu'\rangle \Downarrow v'$ if and only if $\langle P; P', \mu \uplus \mu'\rangle \Downarrow v \uplus v'$. Hence we can recast non-interference as

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

for all memories μ, μ', v, v', 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 (relatively) complete with respect to the operational semantics to provide an alternative characterisation of non-interference. If we use Hoare triples, non-interference can be characterised by

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

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

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

which is easy to show using the rules of Hoare logic. By replacing the $=$-relation by other (partial) equivalence relation, we obtain characterisations of information flow policies that include some forms of declassification. More generally, this kind of characterisation provides us with a way to use existing verification tools to prove, or disprove, information flow policies for a program.

Furthermore, the characterisation 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 is equipped with an appropriate logic. We will illustrate this point by considering a programming language with shared mutable data structures, and using separation logic (Reynolds 2000; Ishtiaq and O'Hearn 2001) to provide a characterisation of non-interference (see Section 8). Second, it can be extended to arbitrary relations between inputs and between outputs, as in, for example, Giacobazzi and Mastroeni (2004a). This more general form of non-interference is useful for providing a characterisation of some controlled forms of declassification, such as delimited information release, which is a form of declassification introduced by Sabelfeld and Myers (2004).

The contributions made by this paper

In this paper we conduct a detailed study of several logical frameworks for characterising non-interference, for both sequential and concurrent non-deterministic programming languages. Our work extends and systematises previous characterisations or criteria for secure information flow policies based on general purpose logics, and allows us to conclude

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

that such logics can be used in an appropriate fashion to provide a criterion for, or even to characterise, 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 for establishing non-interference for languages for which no information flow type system is known – see, in particular, Section 8. The current paper is based on another paper by the authors, which was presented at the IEEE 17th Computer Security Foundations Workshop in June 2004 (Barthe *et al.* 2004). The conference paper is superceded by the present paper in several ways:

— In the new Section 7, we provide a new characterisation of termination-sensitive secure information flow using a weakest precondition calculus.
— We revise our formal framework, correcting some mistakes introduced in Barthe *et al.* (2004). In particular, we revise and formalise the assumptions on the framework of self-composition in the Preliminaries section as well as Fact 1.
— We discuss LTL logic characterisation in Section 9 in greater detail and provide an example program with its characterisation for termination-sensitive and termination-insensitive non-interference.
— We present complete proofs for the main theorems in the paper. We also provide proofs for programs in examples, which can be proved secure with respect to information flow.
— We update the review of related work to take account of the many contributions that have appeared since the publication of Barthe *et al.* (2004).

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, and so on, range over Lang. Let Var be the set of *variables* that may appear in programs, and let x, x', x_1, y, z, and so on, 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.

We assume a set $\mathscr{M}$ of all memories, and let μ, μ', and so on, range over $\mathscr{M}$. In order to define security policies and properties on the programming languages under consideration, we assume two functions: $\mathsf{var} : \mathscr{M} \to \mathsf{Var}$ and an abstraction function $\mathsf{v} : (\mathscr{M} \times \mathsf{Var}) \rightharpoonup \mathscr{V}$, with $\mathscr{V}$ a set of values. The function $\mathsf{var}(\mu)$ 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. (Note that we will not use the 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 memory. Both operations are specified as follows. First, if $\mu \in \mathscr{M}$, $x \in \mathsf{Var}$ and $d \in \mathscr{V}$, then $\mu[x \mapsto d] \in \mathscr{M}$ is some memory such that for all

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

$y \in \mathsf{Var}\ \mathsf{v}(\mu[x \rightrightarrows d], y) = \textbf{if}\ x = y\ \textbf{then}\ d\ \textbf{else}\ \mathsf{v}(\mu, y)$. Note that $\mu[x \rightrightarrows d]$ is one possible variation of μ, though there might be more than one, such that $\mathsf{v}(\mu[x \rightrightarrows d], x) = d$. Thus, if x is a pointer and d is a list, $\mu[x \rightrightarrows d]$ is a modification of possibly many positions in the heap of μ and the assignment of the appropriate location (that is, pointer value) to variable x. Second, if $\mu_1, \mu_2 \in \mathscr{M}$ are two memories satisfying $\mathsf{var}(\mu_1) \cap \mathsf{var}(\mu_2) = \varnothing$, we define $\mu_1 \oplus \mu_2 \in \mathscr{M}$ 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 $\mathsf{v}(\mu_1 \oplus \mu_2, x) = \mathsf{v}(\mu_2, x)$;
— otherwise $\mathsf{v}(\mu_1 \oplus \mu_2, x)$ is undefined.

Notice that $\oplus$ is commutative. We also require that

$$\mathsf{v}(\mu_1[x \rightrightarrows d] \oplus \mu_2, y) = \mathsf{v}(\mu_2, y)$$

for all $x \in \mathsf{var}(\mu_1)$, $y \in \mathsf{var}(\mu_2)$ and $d \in \mathscr{V}$.

Example 1. Consider a language that only manipulates integers, that is, $\mathscr{V} = \mathbb{Z}$. Then $\mathscr{M}$ 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 $\mu[x \rightrightarrows d](y) = \textbf{if}\ x = y\ \textbf{then}\ d\ \textbf{else}\ \mu(y)$.

The operational semantics of the programming language is given by the *transition system* $(\mathsf{Conf}, \rightsquigarrow)$ where $\mathsf{Conf} \subseteq \mathsf{Lang} \times \mathscr{M}$ 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 $\mathsf{var}(S) \subseteq \mathsf{var}(\mu)$). We also use $\rightsquigarrow^*$ to 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 $\mu \in \mathscr{M}$, we have $(\surd, \mu) \not\rightsquigarrow$. On the other hand, we say that a configuration (S, μ) *does not terminate*, denoted by $(S, \mu)\bot$, if the execution of S on memory μ does not terminate (either because of an infinite execution or an abnormal stop, such as a deadlock), that is, $\neg\exists \mu' : (S, \mu) \rightsquigarrow^* (\surd, \mu')$.

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

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

where e is an arithmetic expression and $b, b_0, \ldots, b_n$ are boolean expressions.

The transition relation of Par is defined by the following rules, where memories are the functions of Example 1 and $\mu(e)$, the evaluation of a (boolean or arithmetic) expression e in memory μ, is defined recursively in the usual manner:

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

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

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

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

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

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

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

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

We now state three basic assumptions, which define the scope of application of our self-composition technique. They impose some very general restrictions, which can be viewed as 'health conditions'. Assumptions 1 and 3 are apparently 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 some 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 variables other 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 such that $\mu_1 \oplus \mu_3$ is defined, we have

$$(S, \mu_1 \oplus \mu_3) \rightsquigarrow (S', \mu_1' \oplus \mu_3).$$

Note that this assumption does not prevent object creation: a new object may be created, but it can only be (directly or indirectly) referred to 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

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

we have for all (S', μ_1'),

$$(S, \mu_1) \rightsquigarrow^* (S', \mu_1') \Rightarrow \exists (S', \mu_2') : (S, \mu_2) \rightsquigarrow^* (S', \mu_2')$$

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

and

$$\forall x \in \mathsf{var}(S) : \mathsf{v}(\mu_1', x) = \mathsf{v}(\mu_2', x).$$

Assumption 2 imposes some restrictions on 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 $\mathsf{if}\ (x{=}y) \to S\ [\!]\ (x{\neq}y) \to S'\ \mathsf{fi}$ 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

$$(S[y/x], \mu[y \rightarrowtriangle \mathsf{v}(\mu, x)]) \rightsquigarrow^* (S'[y/x], \mu'[x \rightarrowtriangle d][y \rightarrowtriangle \mathsf{v}(\mu', x)])$$

for some d.

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

The following facts follow from the above assumptions.

Fact 1 (Consequences of the assumptions).

(1) If $\mathsf{var}(S) = \mathsf{var}(\mu_1)$ and $(S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (S', \mu')$, 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 $\mathsf{var}(\mu_1) \cap \mathsf{var}(\mu_3) = \varnothing$.
(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 $(S[y/x], \mu[y \rightarrowtriangle \mathsf{v}(\mu, x)])\bot$.

Items (1) and (2) follow from Assumption 1, and item (3) is a consequence of item (2). Items (4) and (5) are consequences of Assumptions 2 and 3, respectively. It is not difficult to verify that Par satisfies the three assumptions above and has the properties given in Fact 1.

3. A generalisation of non-interference

Let $\phi : \mathsf{Var} \rightharpoonup \mathsf{Var}$ be a partial injective function, which is intended to relate the *variables* of two programs. Let $\mathrm{dom}(\phi) = \{x_1, \ldots, x_n\}$[†] and let $\mathscr{I} \subseteq \mathscr{V}^n \times \mathscr{V}^n$ be a binary relation

[†] We assume variables can always be arranged in a particular order, which we use to arrange the set of variables in tuples.

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

on tuples of values, which is intended to determine the *indistinguishability criterion*. We say that memory μ is $(\phi, \mathscr{I})$-*indistinguishable* from μ', denoted by $\mu \sim_{\phi}^{\mathscr{I}} \mu'$, if

$$\langle (\mathsf{v}(\mu, x_1), .., \mathsf{v}(\mu, x_n)), (\mathsf{v}(\mu', \phi(x_1)), .., \mathsf{v}(\mu', \phi(x_n))) \rangle \in \mathscr{I},$$

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 $\mathscr{I}$.

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 be undefined otherwise. Then $\sim_{\overline{id}_L}$ is the usual indistinguishability relation used to characterise non-interference. It relates memories whose public variables agree in their values, meaning that these memories cannot be distinguished one from each other.

However, our definition of indistinguishability is more flexible. Let $H = \{p\}$ where p is a pointer to a list, and let avrg be the function that computes the average of a list, that is,

$$\mathsf{avrg}([d_1, \ldots, d_N]) = \frac{d_1 + \cdots + d_N}{N}.$$

Let $id_H : \mathsf{Var} \to \mathsf{Var}$ be the identity function on H and be undefined otherwise, and let $\mathscr{A}$ 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 $\sim_{\overline{id}_H}^{\mathscr{A}}$ cannot distinguish between memories μ and μ' that agree on the average value of the list to which p points, that is, that satisfy $\mathsf{avrg}(\mathsf{v}(\mu, p)) = \mathsf{avrg}(\mathsf{v}(\mu', p))$.

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

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

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 the definition of $\sim$. It claims that the relation $\sim_{\phi}^{\mathscr{I}}$ $\sim_{\phi}$ between memories depends only on the values of the variables included in the domain of ϕ.

Proposition 1. For all μ_1, μ_2, μ''_1, μ''_2, $\mathscr{I}$, and $\phi : \mathsf{var}(\mu_1) \to \mathsf{var}(\mu_2)$, we have $\mu_1 \sim_{\phi}^{\mathscr{I}} \mu_2$ if and only if $\mu_1 \oplus \mu''_1 \sim_{\phi}^{\mathscr{I}} \mu_2 \oplus \mu''_2$.

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 $\mathscr{I}$ and $\mathscr{I}'$, which define the relations $\sim_{\phi}^{\mathscr{I}}$ and $\sim_{\phi'}^{\mathscr{I}'}$.

Definition 1.

(1) $S_1 \approx_{\phi', \mathscr{I}'}^{\phi, \mathscr{I}} S_2$ if for all $\mu_1, \mu_2, \mu'_1 \in \mathscr{M}$,

$$\left(\mu_1 \sim_{\phi}^{\mathscr{I}} \mu_2 \ \wedge \ (S_1, \mu_1) \rightsquigarrow^* (\surd, \mu'_1) \right) \Rightarrow \exists \mu'_2 \in \mathscr{M} : (S_2, \mu_2) \rightsquigarrow^* (\surd, \mu'_2) \ \wedge \ \mu'_1 \sim_{\phi'}^{\mathscr{I}'} \mu'_2.$$

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

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

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

(3) Let $\mathscr{I}, \mathscr{I}' \subseteq \mathscr{V}^n \times \mathscr{V}^n$ with $n = \#\,\mathsf{var}(S)$.

(a) S is *termination-sensitive (TS)* $(\mathscr{I}, \mathscr{I}')$*-secure* if and only if $S \approx^{id,\mathscr{I}}_{id,\mathscr{I}'} S$.

(b) S is *termination-insensitive (TI)* $(\mathscr{I}, \mathscr{I}')$*-secure* if and only if $S \approx^{id,\mathscr{I}}_{id,\mathscr{I}'} S$.

Informally, $S_1 \approx^{\phi,\mathscr{I}}_{\phi',\mathscr{I}'} S_2$ holds (in words, 'S_1 is termination sensitive and non-interfering with S_2') if for any two input indistinguishable memories, a 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. $S_1 \approx^{\phi,\mathscr{I}}_{\phi',\mathscr{I}'} S_2$ (in words, 'S_1 is termination insensitive and non-interfering with program S_2') is a weaker concept in the sense that S_2 might diverge. Finally, a program is (TS or TI) $(\mathscr{I}, \mathscr{I}')$-secure if it is (TS or TI) non-interfering with itself.

Traditional non-interference is characterised 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 appearing in the literature (see, for example, Goguen and Meseguer (1982), Volpano *et al.* (1996), Smith and Volpano (1998) and Joshi and Leino (2000)).

However, our definitions are more flexible than the usual formulations of non-interference. Indeed, the latter usually require that executions from indistinguishable memories end with 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 $(\phi, \mathscr{I})$-indistinguishability and output indistinguishability with $(\phi', \mathscr{I}')$-indistinguishability.

4. Information flow using composition and renaming

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

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

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

The operation $\triangleright$ is the first of two foundations on which our result is built. Notice that non-interference, as given by Definition 1, considers the executions of programs S_1 and S_2 separately. By composing $S_1 \triangleright S_2$, properties (a) and (b) above allow us to put these executions one after the other. Therefore, we can give another characterisation of security.

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

Definition 2. Let S_1, S_2 be two programs such that $\mathsf{var}(S_1) \cap \mathsf{var}(S_2) = \varnothing$. We define $S_1 \stackrel{\triangleright}{\approx}{}^{\phi,\mathscr{I}}_{\phi',\mathscr{I}'} S_2$ (and $S_1 \stackrel{\triangleright}{\sim}{}^{\phi,\mathscr{I}}_{\phi',\mathscr{I}'} S_2$ 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 $\mathsf{var}(\mu_2) = \mathsf{var}(S_2)$, we have

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

Note that this definition has the same shape as Definition 1. However, while

$$S_1 \stackrel{}{\approx}{}^{\phi,\mathscr{I}}_{\phi',\mathscr{I}'} S_2$$

considers executions of two different programs (S_1 and S_2),

$$S_1 \stackrel{\triangleright}{\approx}{}^{\phi,\mathscr{I}}_{\phi',\mathscr{I}'} S_2$$

considers the execution of only one program ($S_1 \triangleright S_2$): the first part of the execution (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, the non-interference of two programs can be viewed as the non-interference of a single program (namely, the composition of those two programs).

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

(a) $S_1 \stackrel{}{\approx}{}^{\phi,\mathscr{I}}_{\phi',\mathscr{I}'} S_2$ if and only if $S_1 \stackrel{\triangleright}{\approx}{}^{\phi,\mathscr{I}}_{\phi',\mathscr{I}'} S_2$; and

(b) $S_1 \stackrel{}{\sim}{}^{\phi,\mathscr{I}}_{\phi',\mathscr{I}'} S_2$ if and only if $S_1 \stackrel{\triangleright}{\sim}{}^{\phi,\mathscr{I}}_{\phi',\mathscr{I}'} S_2$.

Proof.

(a) *Termination-sensitive case:*

By Proposition 1 and the commutativity of $\oplus$, we conclude that $\mu_1 \sim^{\mathscr{I}}_{\phi} \mu_2$ if and only if

$$\mu_1 \oplus \mu_2 \sim^{\mathscr{I}}_{\phi} \mu_1 \oplus \mu_2.$$

By Fact 1 (2), we have $(S_1, \mu_1) \rightsquigarrow^* (\surd, \mu_1')$ if and only if

$$(S_1, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2),$$

and by the definition of $\triangleright$, we have

$$(S_1, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2)$$

if and only if

$$(S_1 \triangleright S_2, \mu_1 \oplus \mu_2) \rightsquigarrow^* (S_2, \mu_1' \oplus \mu_2).$$

Using similar arguments, we can conclude that

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

from which (a) follows.

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

(b) *Termination-insensitive case:*

It follows by (1) and Fact 1 (3) that

$$\begin{array}{l}\left(\exists \mu_2' : (S_2,\mu_2) \rightsquigarrow^* (\surd,\mu_2')\ \wedge\ \mu_1' \sim^{\mathscr{I}'}_{\phi'} \mu_2'\right) \vee (S_2,\mu_1')\bot \\ \text{iff}\ \left(\exists \mu_2' : (S_2,\mu_1'\oplus\mu_2) \rightsquigarrow^* (\surd,\mu_1'\oplus\mu_2')\ \wedge\ \mu_1'\oplus\mu_2' \sim^{\mathscr{I}'}_{\phi'} \mu_1'\oplus\mu_2'\right) \vee (S_2,\mu_1'\oplus\mu_2)\bot\end{array}$$

The result then follows from the hypothesis equivalence shown in case (a). □

Programs sharing variable names are not handled by Definition 2 (or Theorem 1). Through variable renaming, which is the second of our foundations, 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, which 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

$$[y/x](z) = \textbf{if}\ z \neq x\ \textbf{then}\ z\ \textbf{else}\ y.$$

Then:

(a) $S_1 \approx^{\phi,\mathscr{I}}_{\phi',\mathscr{I}'} S_2$ if and only if $S_1 \approx^{[y/x]\phi,\mathscr{I}}_{[y/x]\phi',\mathscr{I}'} S_2[y/x]$

(b) $S_1 \lessapprox^{\phi,\mathscr{I}}_{\phi',\mathscr{I}'} S_2$ if and only if $S_1 \lessapprox^{[y/x]\phi,\mathscr{I}}_{[y/x]\phi',\mathscr{I}'} S_2[y/x]$,

where $S_2[y/x]$ is program S_2 with variable x renamed to y, and $[y/x]\phi$ is a shorthand for $[y/x] \circ \phi$.

Proof.

(a) We consider the if and only if cases separately:

($\Rightarrow$) Let $\mu_1 \sim^{\mathscr{I}}_{[y/x]\phi} \mu_2$ and $(S_1,\mu_1) \rightsquigarrow^* (\surd,\mu_1')$. Note that

$$\forall z \in \mathrm{dom}(\phi) : \mathsf{v}(\mu_2,[y/x]\phi(z)) = \mathsf{v}(\mu_2[x \rightmapsto \mathsf{v}(\mu_2,y)],\phi(z)),$$

so $\mu_1 \sim^{\mathscr{I}}_{\phi} \mu_2[x \rightmapsto \mathsf{v}(\mu_2,y)]$. As a consequence, since $S_1 \approx^{\phi,\mathscr{I}}_{\phi',\mathscr{I}'} S_2$, there is $\mu_2' \in \mathscr{M}$ such that

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

By Assumption 3, there is some d such that

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

Note that since $x \notin \mathsf{var}(S_2[y/x])$, we have

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

for all $w \in \mathsf{var}(S_2[y/x])$. Hence, by Assumption 2,

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

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

for some μ_2'' such that

$$\forall w \in \mathsf{var}(S_2[y/x]) : \mathsf{v}(\mu_2'[x \mapsto d][y \mapsto \mathsf{v}(\mu_2', x)], w) = \mathsf{v}(\mu_2'', w).$$

Note also 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 $\mu_1' \sim_{\phi'}^{\mathscr{I}'} \mu_2'$, we finally have $\mu_1' \sim_{[y/x]\phi'}^{\mathscr{I}'} \mu_2''$.

($\Leftarrow$) It is clear that $x \notin \mathsf{var}(S_2[y/x])$ and, for all $z \in \mathrm{dom}(\phi)$, we have (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

$$S_1 \approx_{[y/x]\phi', \mathscr{I}}^{[y/x]\phi, \mathscr{I}} S_2[y/x]$$

implies

$$S_1 \approx_{\phi', \mathscr{I}'}^{\phi, \mathscr{I}} S_2[y/x][x/y].$$

Hence,

$$S_1 \approx_{\phi', \mathscr{I}'}^{\phi, \mathscr{I}} S_2.$$

(b) ($\Rightarrow$) For the case of TI non-interference, we take over (2) from the proof of part (a), but for this part suppose instead that $(S_2, \mu_2[x \mapsto \mathsf{v}(\mu_2, y)])\perp$. By Fact 1 (5),

$$(S_2[y/x], \mu_2[x \mapsto \mathsf{v}(\mu_2, y)][y \mapsto \mathsf{v}(\mu_2[x \mapsto \mathsf{v}(\mu_2, y)], x)])\perp.$$

Taking into account equation (3) above, by Fact 1 (4), $(S_2[y/x], \mu_2)\perp$, which, together with the previous case, proves ($\Rightarrow$).

($\Leftarrow$) We use the same reasoning for this as in ($\Leftarrow$) for (a). □

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

(a) $S_1 \approx_{\phi', \mathscr{I}'}^{\phi, \mathscr{I}} S_2$ if and only if $S_1 \approx_{\xi \circ \phi', \mathscr{I}'}^{\xi \circ \phi, \mathscr{I}} S_2[\xi]$

(b) $S_1 \approx_{\phi', \mathscr{I}'}^{\phi, \mathscr{I}} S_2$ if and only if $S_1 \approx_{\xi \circ \phi', \mathscr{I}'}^{\xi \circ \phi, \mathscr{I}} S_2[\xi]$,

where $S_2[\xi]$ is program S_2 whose variables have been renamed according to function ξ.

Proof. We use induction on the number of variables x such that $\xi(x) \neq x$.

— Case $n = 0$:
This sorresponds to the identity and is trivial.

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

— Case $n \geqslant 1$:
We proceed by induction using Lemma 1.
We will only give the proof for part (a) since the proof of part (b) follows along similar lines. Let $x \in \mathsf{var}(S_2)$ and ξ be such that $\xi(x) = x$. Let y be a fresh variable not in the image of ξ. Note that the number of variables z such that $[y/x]\xi(z) = z$ is exactly one more than those such that $\xi(z) = z$. Now we have

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

□

Putting Theorems 1 and 2 together, we have the following corollary.

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

(1) S is TS (respectively, TI) $(\mathscr{I}, \mathscr{I}')$-secure.
(2) $S \approx^{\xi,\mathscr{I}}_{\xi,\mathscr{I}'} S[\xi]$ (respectively, $S \approx^{\xi,\mathscr{I}}_{\xi,\mathscr{I}'} S[\xi]$).
(3) $S \overset{\triangleright}{\approx}{}^{\xi,\mathscr{I}}_{\xi,\mathscr{I}'} S[\xi]$ (respectively, $S \overset{\triangleright}{\sim}{}^{\xi,\mathscr{I}}_{\xi,\mathscr{I}'} S[\xi]$).

Corollary 1 allows us to check whether a program S is secure by analysing single executions of the program $S \triangleright S[\xi]$. But this is what verification logics are used for. We will characterise $(\mathscr{I}, \mathscr{I}')$-security in some of these logics.

5. Deterministic programs

Simpler definitions for non-interference can be obtained if the program S under consideration 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, then, for all μ, either $(S, \mu)\perp$ 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 intermediate points in the program but instead consider only complete executions. This allows us to check security by simply analysing 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 $\mathsf{var}(S)'$ be as in Corollary 1.

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

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

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

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

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

The proof of Theorem 3 is given 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 its 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 $x \in \mathsf{Var}$. Note, 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 $(S, \mu)\perp$ or for all $\mu_1, \mu_2 \in \{\mu' \mid (S, \mu) \rightsquigarrow^* (\surd, \mu')\}$, we have $\mathsf{v}(\mu_1, x) = \mathsf{v}(\mu_2, x)$ for every $x \in \mathsf{Var}$.

The proof of the next theorem closely follows 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 $\mathsf{var}(S)'$ be 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 programs 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 the 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 will be central in the next two sections. Note that Theorem 3 requires that memory should be separable by the 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.

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

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 $\mathsf{var}(S)'$ be as in Corollary 1. Then:

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

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

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

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

6. Hoare logic

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

Let While be the subset of Par that does not contain parallel composition and limits 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 be a While program. The meaning of a Hoare triple (Hoare 1969) $\{P\}\, S\, \{Q\}$ is 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:

$$\{P[e/x]\}\ x := e\ \{P\}$$

$$\frac{P' \Rightarrow P \quad \{P\}\, S\, \{Q\} \quad Q \Rightarrow Q'}{\{P'\}\, S\, \{Q'\}}$$

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

$$\frac{\{P\}\, S_1\, \{R\} \quad \{R\}\, S_2\, \{Q\}}{\{P\}\, S_1\, ;\, S_2\, \{Q\}}$$

$$\frac{\{P \wedge b\}\, S\, \{P\}}{\{P\}\ \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}\ \{P \wedge \neg b\}}$$

Hoare logic is sound and (relatively) complete with respect to the operational semantics (Cook 1978). That is, for all program S and predicates P and Q, the assertion $\{P\}\, S\, \{Q\}$ is provable if and only if for all μ, μ', we have $\mu \models P$ and $(S, \mu) \rightsquigarrow^* (\surd, \mu')$ imply $\mu' \models Q$, where $\mu \models P$ means that P holds whenever every program variable x appearing in P is replaced by the value $\mathsf{v}(\mu, x)$.

Suppose $\mathbf{I}(\mathscr{I})$ is a first-order predicate representing the indistinguishability criterion $\mathscr{I}$ on the values of the program, that is,

$$\mu \models \mathbf{I}(\mathscr{I}) \quad \text{if and only if} \quad \mu \sim_{\xi}^{\mathscr{I}} \mu \quad \left(\text{if and only if} \quad (\mathsf{v}(\mu, \vec{x}), \mathsf{v}(\mu, \vec{x}')) \in \mathscr{I}\right)$$

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

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

(a)

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

(b)

Fig. 1. Security proofs in Hoare logic

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 $\mathbf{I}(\mathscr{I})$ 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 $(\mathscr{I}_1, \mathscr{I}_2)$-security can be characterised in Hoare logic as follows:

$$S \text{ is TI } (\mathscr{I}_1, \mathscr{I}_2)\text{-secure if and only if } \{\mathbf{I}(\mathscr{I}_1)\}\, S\, ;\, S[\xi]\, \{\mathbf{I}(\mathscr{I}_2)\} \text{ is provable.}$$

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

S is TI $(\mathscr{I}_1, \mathscr{I}_2)$-secure if and only if (by Corollary 2 (2))

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

if and only if (by the definition of $\mathbf{I}$)

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

if and only if (by soundness and completeness, provided $\mathbf{I}$ is definable)

$$\{\mathbf{I}(\mathscr{I}_1)\}\, S\, ;\, S[\xi]\, \{\mathbf{I}(\mathscr{I}_2)\}$$

is provable □

Example 5. Let x_l and y_h be public and confidential variables, respectively, in the program $x_l := x_l + y_h\, ;\, x_l := x_l - y_h$. We show that it is non-interfering. Indistinguishability in this case is characterised by the predicate $\mathbf{I}(=_{\{x_l\}}) \equiv (x_l = x_l')$. The proof is given in Figure 1(a).

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

The generality of our definition is useful for providing a characterisation of some forms of controlled declassification. Declassification allows us to leak some confidential information without being too revealing. A semantic characterisation of this kind of property has been given in Sabelfeld and Myers (2004), where it was called *delimited release*. A typical example is a program S that reports 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, as in the following example.

Example 6. PIN access control provides an example involving declassification. In the program

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

the 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 preventing access to the system, is a public output variable. The declassified information should only reveal whether the input number (in) agrees with the PIN number (pin) or not, and such information is revealed by granting or preventing access (indicated in acc). So we require the program to be $(\mathscr{I}, =_{\{acc\}})$-secure, where $\mathscr{I}$ is such that $\sim_{id}^{\mathscr{I}}$ if and only if

$$(\mu(in) = \mu(pin)) \Leftrightarrow (\mu'(in) = \mu'(pin)).$$

Hence,

$$\mathbf{I}(\mathscr{I}) \equiv ((in = pin) \leftrightarrow (in' = pin'))$$

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 characterisation of termination-sensitive $(\mathscr{I}, \mathscr{I}')$-security for deterministic programs, where one needs to ensure that if S terminates for some memory μ_1, then $S[\xi]$ terminates for an 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 (wp) (Dijkstra 1997).

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

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

In particular,

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

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

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

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

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

where

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

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

S is TS $(\mathscr{I}_1, \mathscr{I}_2)$-secure if and only if $\mathbf{I}(\mathscr{I}_1) \wedge wp(S, \textsf{true}) \Rightarrow wp(S ; S[\xi], \mathbf{I}(\mathscr{I}_2))$.

Proof. By Corollary 2 (1), S is TS $(\mathscr{I}_1, \mathscr{I}_2)$-secure if and only if

$$\begin{aligned} \forall \mu : \big(\big(\mu \sim_{\xi}^{\mathscr{I}_1} \mu \ \wedge\ \exists \mu' : (S, \mu) \rightsquigarrow^* (\surd, \mu') \big) \\ \Rightarrow \big(\exists \mu'' : (S ; S[\xi], \mu) \rightsquigarrow^* (\surd, \mu'') \ \wedge\ \mu'' \sim_{\xi}^{\mathscr{I}_2} \mu'' \big) \big) \end{aligned}$$

if and only if (by the definition of $\mathbf{I}$)

$$\begin{aligned} \forall \mu : \big(\big(\mu \models \mathbf{I}(\mathscr{I}_1) \ \wedge\ \exists \mu' : (S, \mu) \rightsquigarrow^* (\surd, \mu') \big) \\ \Rightarrow \big(\exists \mu'' : (S ; S[\xi], \mu) \rightsquigarrow^* (\surd, \mu'') \ \wedge\ \mu'' \models \mathbf{I}(\mathscr{I}_2) \big) \big) \end{aligned}$$

if and only if (by (5))

$$\begin{aligned} \forall \mu : \big(\big(\mu \models \mathbf{I}(\mathscr{I}_1) \ \wedge\ \mu \models wp(S, \textsf{true}) \big) \\ \Rightarrow \big(\exists \mu'' : (S ; S[\xi]), \mu) \rightsquigarrow^* (\surd, \mu'') \ \wedge\ \mu'' \models \mathbf{I}(\mathscr{I}_2) \big) \big) \end{aligned}$$

if and only if (by (4))

$$\big(\mathbf{I}(\mathscr{I}_1) \ \wedge\ wp(S, \textsf{true}) \big) \ \Rightarrow\ wp(S ; S[\xi], \mathbf{I}(\mathscr{I}_2)).$$

□

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

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

Example 7. Consider the following 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 $wp(S, \mathsf{true})$:

$$\exists k : k \geqslant 0 : y \geqslant 3 \vee (y \leqslant 3 \wedge y \geqslant 1 \wedge y \geqslant 3 - k),$$

which is equivalent to $y \geqslant 1$.

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

Hoare logic with total correctness and wp are related by $P \Rightarrow wp(S, Q)$ if and only if $[P]\,S\,[Q]$, where $[P]\,S\,[Q]$ denotes the Hoare triple for total correctness. Following Proposition 3, a first attempt to characterise $(\mathscr{I}_1, \mathscr{I}_2)$-security using Hoare logic with total correctness yields

$$[\mathbf{I}(\mathscr{I}_1) \wedge wp(S, \mathsf{true})]\,S\,;S[\xi]\,[\mathbf{I}(\mathscr{I}_2)]\,.$$

However, this characterisation 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 characterisation using Hoare logic with total correctness is not only impossible in its pure form, but also requires a second-order quantification. In fact, the characterisation should be written as follows:

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

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

8. Separation logic

Separation logic is an extension of Hoare logic for reasoning about shared mutable data structures (Ishtiaq and O'Hearn 2001; Reynolds 2002). 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). The command $x := \mathsf{cons}(e_1, e_2)$ 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 to by e. (Binary tuples are sufficient for our purposes, though arbitrary n-tuples appear in the literature and could also be

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

considered here.) So, $x := e.i$ and $x.i := e$, respectively, allow us to read and update the heap. 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 $\mathscr{V} = \mathbb{Z} \cup \mathsf{Loc}$, then $\mathscr{S} = \mathsf{Var} \to \mathscr{V}$ is the set of *stores* and $\mathscr{H} = \mathsf{Loc} - \{\mathsf{nil}\} \to (\mathscr{V} \times \mathsf{Loc})$ is the set of *heaps*. As a consequence, variables can have type $\mathbb{Z}$ or type Loc. Finally, $\mathscr{M} = \mathscr{S} \times \mathscr{H}$.

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, which holds if the heap is empty; and
— $P * Q$, which is called *separating conjunction*, holds if the heap can be split into two parts, one satisfying P and the other Q.

There exists a calculus for these operations, which also includes the separating implication $P \mathrel{-\!\!*} Q$ – see Ishtiaq and O'Hearn (2001) and Reynolds (2000). The meaning of an assertion depends on both the store and the heap:

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

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*, which allows us 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 as follows (we omit the 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 or 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 either e or e_2, then

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

— Finally,

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

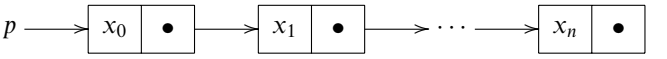
Using separation logic we can define inductive predicates that refer to structures in the heap (see Reynolds (2000) and Ishtiaq and O'Hearn (2001)). For simplicity, we will only

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

consider the predicate list, which is defined by

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

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



As we have already 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 $s, s_1, s_2 \in \mathscr{S}$, $h, h_1, h_2 \in \mathscr{H}$ and $x \in \mathsf{Var}$, we define $\mathsf{var}(s, h) = \mathrm{dom}(s)$ by

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

where $\overline{\mathsf{v}}(h, l)$ returns the list pointed to by l, that is,

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

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

$$l' \in \mathrm{reach}(l,h) = \{(\mathrm{snd} \circ h)^k(l) \mid k \geqslant 0\}.$$

Hence, the *set of all locations reachable from store s in h* is defined by

$$\mathrm{reach}(s,h) = \bigcup \{\mathrm{reach}(l,h) \mid l \in \mathrm{ran}(s) \cap \mathsf{Loc} - \{\mathsf{nil}\}\}.$$

So (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 $x \in \mathsf{var}(s_1, h_1)$, we have $\mathsf{v}((s_1 \oplus s_2, h_1 \oplus h_2), x)$ may be defined when $\mathsf{v}((s_1, h_1), x)$ is not (and hence not satisfying the requirement of $\oplus$ in Section 2).

Now that v and $\overline{\mathsf{v}}$ are defined, note 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 the variables in S that have type Loc (the pointer variables) and $\{y_1, .., y_m\}$ be all the 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 $\vec{y}'$. We use the denotations $\vec{xs} = (xs_1, .., xs_n)$ and $\vec{xs}' = (xs_1', .., xs_n')$. We fix this notation for the rest of this section.

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

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

$$\begin{aligned} \exists \vec{xs}, \vec{xs}' : \big(&\big(\textstyle\bigwedge_{1 \leqslant i \leqslant n} (\mathsf{list}.xs_i.x_i * \mathsf{true})\big) * \big(\textstyle\bigwedge_{1 \leqslant i \leqslant n} (\mathsf{list}.xs_i'.x_i' * \mathsf{true})\big) \big) \\ &\wedge \mathbf{I}_v(\vec{xs}, \vec{xs}', \mathscr{I}) \end{aligned}$$

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

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

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

where $\mathsf{v}(\mu, \vec{y})$ is defined as in Section 6, and $\vec{ds}$ and $\vec{ds}'$ are actual list values. Note that the following holds:

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

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

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

Therefore, $\mathbf{I}_{sl}(\mathscr{I})$ has two parts: the first part states the separation of the heap identifying the list values represented by the pointer variables, and the second states 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 (Ishtiaq and O'Hearn 2001). As a consequence, security in separation logic can be completely characterised by the following proposition.

Proposition 4. S is TI $(\mathscr{I}_1, \mathscr{I}_2)$-secure if and only if $\{\mathbf{I}_{sl}(\mathscr{I}_1)\}\ S\ ;\ S[\xi]\ \{\mathbf{I}_{sl}(\mathscr{I}_2)\}$ is provable.

Before continuing with the proof we state the following property.

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, the 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, the value of x_l will be 0 or 1, revealing whether secret information has been allocated before.

Property 1 follows by structural induction using the operational rules defined in Ishtiaq and O'Hearn (2001). Using the previous observations, we can now finally proceed to prove the correctness and completeness of the characterisation in Separation Logic.

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

Proof. We first prove that $\mathbf{I}_{sl}(\mathscr{I})$ characterises indistinguishability in a separable memory, that is, 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, we observe that $\overline{\mathsf{v}}(h, l)$ is defined if and only if $\mathrm{reach}(l, h) \subseteq \mathrm{dom}(h)$.

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 $\mathrm{dom}(s) = \mathsf{var}(S) \cup \mathsf{var}(S)'$, then

$$\exists h_1, h_2 : h = h_1 \oplus h_2 : \overline{\mathsf{v}}(h_1, s(\vec{x})) \text{ and } \overline{\mathsf{v}}(h_2, s(\vec{x}')) \text{ are defined}$$

if and only if (since s is a function with $\mathrm{dom}(s) = \mathsf{var}(S) \cup \mathsf{var}(S)'$)

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

if and only if (from the observation above)

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

if and only if (by the definition 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 $\mathbf{I}_{sl}(\mathscr{I})$. We have

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

if and only if (by unfolding $\mathbf{I}_{sl}(\mathscr{I})$)

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

if and only if (by semantics (equation (8)))

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

if and only if (by the definition of $\mathbf{I}_v$ and equality on vectors)

$$\begin{aligned} \exists \vec{ds}, \vec{ds}' : \exists h_1, h_2 : h = h_1 \oplus h_2 : \vec{ds} = \mathsf{v}((s, h_1), \vec{x}) \ \wedge\ \vec{ds}' = \mathsf{v}((s, h'_2), \vec{x}') \\ \wedge\ (\langle \mathsf{v}(\mu, \vec{y}), \vec{ds} \rangle, \langle \mathsf{v}(\mu, \vec{y}'), \vec{ds}' \rangle) \in \mathscr{I} \end{aligned}$$

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

if and only if (since $\overline{\mathsf{v}}(h_1, s(x_i))=\mathsf{v}((s,h),x_i)=ds_i$ and $\overline{\mathsf{v}}(h_2, s(x_i'))=\mathsf{v}((s,h),x_i') = ds_i'$, for $1 \leqslant i \leqslant n$)

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

if and only if (since $\exists v : f(z) = v$ if and only if $f(z)$ is defined, and the definition of $\sim_\xi^{\mathscr{I}}$)

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

if and only if (by (9))

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

Now S is TI $(\mathscr{I}_1, \mathscr{I}_2)$-secure
if and only if (by Property 1 and Theorem 4)

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

if and only if (logic)

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

if and only if (by logic and (10))

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

if and only if (since Separation Logic is sound and complete)

$$\{\mathbf{I}_{sl}(\mathscr{I}_1)\} \ S \ ; S[\xi] \ \{\mathbf{I}_{sl}(\mathscr{I}_2)\}$$

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 ≠ 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
```

The variables s_{aux} and p_{aux} are specifically included to meet the syntax restrictions imposed on the language. We call this program AV_SAL (for 'AVerage SALary').

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

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 $\mathscr{A}$ 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 $(\mathscr{A}, =_{\{a_l\}})$-secure. Thus, the precondition $\mathbf{I}_{sl}(\mathscr{A})$ and postcondition $\mathbf{I}_{sl}(=_{\{a_l\}})$ predicates are

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

and

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

respectively, where we have used the notation $|ps|$ for the length of list ps, and $\sum ps$ for the sum of all numbers in ps with $\sum[\,] = 0$. We assume that the heap contains exactly the lists pointed to by *lsalaries* and *lsalaries*′, so we can omit '$* \mathsf{true}$' in these definitions.

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

The *Relational Separation Logic* (Yang 2007) is closely related to Separation Logic. Relational separation logic is a logic to specify relations between two pointer programs and prove their specifications. Note that TS-$(\mathscr{I}_1, \mathscr{I}_2)$-security can also be expressed 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 this 'quadruple' can hold both the program and its renamed copy separately.

9. Temporal logics

Computation Tree Logic (CTL for short) (Clarke *et al.* 1986) is a temporal logic that extends propositional logic with modalities to express properties on the branching structure of a non-deterministic execution. That is, CTL temporal operators allow us to quantify over execution paths (that is, maximal transition sequences leaving a particular state). Apart from the usual propositional operations (atomic propositions, $\neg$, $\vee$, $\wedge$, $\rightarrow$,...), CTL provides (unary) temporal operators EF, AF, EG and AG. Formula $\mathsf{EF}\,\phi$ states that there *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 there *exists* an execution path in which ϕ *globally* holds (that is, 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 assigns a set of atomic propositions to each configuration in Conf, so $\mathsf{Prop}(c)$ 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

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

$c_n \not\rightsquigarrow$. For $i \geqslant 0$, let $\rho_i = c_i$ be the ith state in ρ (if ρ is finite, $i+1$ must not exceed ρ's length).

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

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

AG and EG are the duals 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, we have $c \models p$ if and only if $p \in \mathsf{Prop}(c)$. The semantics of the propositional operators $\neg$, $\wedge$, $\vee$ and $\rightarrow$ are as usual (for example, $c \models \phi \wedge \psi$ if and only if $c \models \phi$ and $c \models \psi$).

In this section, we impose an extra requirement on the composition $S_1 \triangleright S_2$, which allows us to identify syntactically the moment of the execution at 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 helps us keep the CTL formulas that characterise security simple.

Let end be the atomic proposition that indicates that the execution reaches a successfully terminating state, that is, $\mathsf{end} \in \mathsf{Prop}(S, \mu)$ if and only if $S = \surd$. Let mid indicate that program $S[\xi]$ is about to be executed, that is, $\mathsf{mid} \in \mathsf{Prop}(S', \mu)$ if and only if $S' = S[\xi]$. Let $\mathsf{Ind}[\mathscr{I}]$ be an atomic proposition indicating indistinguishability in a state. Thus $\mathsf{Ind}[\mathscr{I}] \in \mathsf{Prop}(S, \mu)$ if and only if $\mu \sim_\xi^{\mathscr{I}} \mu$. We use $S \models \Phi$ to denote $\forall \mu : (S, \mu) \models \Phi$. For simplicity, we consider simple memories as in Example 1 (more complex states are possible, but we would need to introduce additional atomic propositions to characterise 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 $(\mathscr{I}_1, \mathscr{I}_2)$-secure if and only if $S \triangleright S[\xi]$ satisfies

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

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 characterisation of TS security given by Proposition 5 can be proved using Corollary 1 (see Appendix C).

Requirement (c) is necessary so that formula (11) is not confused by the satisfaction of mid at 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 sequential composition alone:

```
while x < 2 do
    if  [] x = 0 →  x := 2
        [] true →   x := 1   fi
od
```

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

Note 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 executed 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, an intermediate skip instruction (or a trivial assignment): $S^W; \mathsf{skip}; S^W[\xi]$. 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 note 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 characterisation in CTL is obtained from (11) by allowing non-termination as follows.

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

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

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.

Example 9. Let y_h be a confidential variable in the following programs (which are borrowed from Joshi and Leino (2000)):

(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 can check whether they are non-interfering (Smith and Volpano 1998; Joshi and Leino 2000), that is, whether they are $(=_L, =_L)$-secure. We use CTL, and for this we set $\mathsf{Ind}[=_\varnothing] \equiv \mathsf{true}$. The automaton of the (self-composed) programs (a) and (b) are shown in Figure 2. In the picture, variables can only take the value 0 or 1. Also, states are represented by tuples (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 identified by a small incoming arrow.

Observe that both programs satisfy the TI formula

$$\mathsf{true} \rightarrow \mathsf{AG}(\mathsf{mid} \rightarrow ((\mathsf{AG}\,\neg\mathsf{end}) \vee \mathsf{EF}(\mathsf{end} \wedge \mathsf{true}))).$$

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

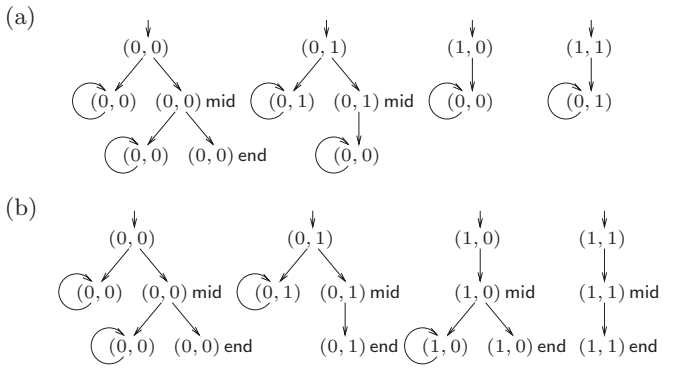


Fig. 2. Automata for the programs of Example 9

As observed in Joshi and Leino (2000), 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 this leakage. Notice that the second automaton from the left has an execution that completes its 'first phase' but never terminates. However, the formula is valid in program (b).

Linear Temporal Logic characterisation of security.

A similar characterisation can be given for Linear Temporal Logic (LTL) (Manna and Pnueli 1992) 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*, that is, in some suffix of this execution. $\mathsf{G}\,\phi$ holds in a program execution if ϕ holds *globally*, that is, in all suffixes of this execution.

In a deterministic setting, the semantics of F and G can be characterised in terms of reachability:

$$c \models \mathsf{F}\,\phi \text{ if and only if } (\exists c' : c \rightsquigarrow^* c' : c' \models \phi)$$

and

$$c \models \mathsf{G}\,\phi \text{ if and only if } (\forall c' : c \rightsquigarrow^* c' : c' \models \phi).$$

Using Corollary 2, TS and TI $(\mathscr{I}_1, \mathscr{I}_2)$-security can be characterised in LTL by the formulas

$$\mathsf{Ind}[\mathscr{I}_1] \rightarrow ((\mathsf{F}\,\mathsf{mid}) \rightarrow \mathsf{F}(\mathsf{end} \wedge \mathsf{Ind}[\mathscr{I}_2]))$$

and

$$\mathsf{Ind}[\mathscr{I}_1] \rightarrow \mathsf{G}(\mathsf{end} \rightarrow \mathsf{Ind}[\mathscr{I}_2]),$$

respectively.

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

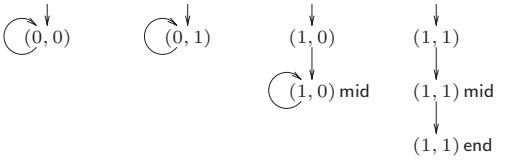


Fig. 3. Automata for program of Example 10

It is known that CTL and LTL are incomparable in expressiveness: $\mathsf{AG}(\phi \rightarrow \mathsf{EF}\,\psi)$ is a typical CTL formula that cannot be expressed in LTL; it can be shown that this is also true for $\mathsf{AG}(\phi \rightarrow (\mathsf{AG}\,\psi \vee \mathsf{EF}\,\psi))$. These formulas occur as non-trivial subformulas of the CTL characterisations of security. As a consequence, security in a non-deterministic setting cannot be characterised using LTL (at least not using our technique).

Example 10. Let y_h be a confidential variable in the program $\mathsf{while}\ y_h{=}0\ \mathsf{do}\ y_h := 0\ \mathsf{od}$. We will check non-interference, that is, $(=_L, =_L)$-security. We have $\mathsf{Ind}(=_\varnothing) \equiv \mathsf{true}$ 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 only take the values 0 or 1. As before, a state is represented by a tuple (d, d') containing the values of y_h and y'_h, respectively, and the validity of mid and end is shown next to the state. Note 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 (Joshi and Leino 2000). A sharp adversary that can observe *possibilistic* non-termination may detect that a possible execution of the same instance of a program (that is, running with the same starting memory) stalls indefinitely. Such an adversary can observe a difference between program (b) under $y_h = 0$ (which sometimes terminates, but sometimes does not) or under $y_h = 1$ (which always terminates). To this extent, our characterisation of TS $(\mathscr{I}_1, \mathscr{I}_2)$-security fails.

So far, we have considered *strict* non-termination: $(S, \mu)\perp$ denotes the fact that S *does not* terminate in μ. A notion of *possibilistic* non-termination can also be given: let $(S, \mu)\nearrow$ denote the fact that there is an execution of S from memory μ that does not terminate. That is, $(S, \mu)\nearrow$ if and only if there exists ρ such that $(S, \mu){=}\rho_0$ and $\forall i : i \geqslant 0 : \neg\exists \mu' : \rho_i{=}(\surd, \mu')$.

From Definition 1, S is (TS) $(\mathscr{I}_1, \mathscr{I}_2)$-secure if for all μ_1, μ_2 such that $\mu_1 \sim_{id}^{\mathscr{I}_1} \mu_2$,

(0) $\quad \forall \mu'_1 : (S, \mu_1) \rightsquigarrow^* (\surd, \mu'_1) \Rightarrow (\exists \mu'_2 : (S, \mu_2) \rightsquigarrow^* (\surd, \mu'_2) \wedge \mu'_1 \sim_{id}^{\mathscr{I}_2} \mu'_2)$.

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

(i) $(S, \mu_1)\perp \Rightarrow (S, \mu_2)\perp$.
(ii) $(S, \mu_1)\perp \Rightarrow (S, \mu_2)\nearrow$.

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

(iii) $(S,\mu_1)\nearrow \Rightarrow (S,\mu_2)\perp$.
(iv) $(S,\mu_1)\nearrow \Rightarrow (S,\mu_2)\nearrow$.

Since $\neg(S,\mu)\perp$ if and only if $\exists\mu' : (S,\mu) \rightsquigarrow^* (\surd,\mu')$, and provided $\mathscr{I}_1$ is symmetric, (i) can be deduced from (0). Since (0) implies (i), and $(S,\mu)\perp$ implies $(S,\mu)\nearrow$, it follows that (ii) is also redundant.

Condition (iii) states that if a program *may* not terminate then it *must* not terminate in any indistinguishable state. As a consequence, it considers that any program that sometimes terminates and sometimes does not is insecure. In particular, program (b) in Example 9 is insecure under this condition. But so is

$$\mathsf{if}\ [\!]\ \mathsf{true} \to \mathsf{while\ true\ do}\ h := h\ \mathsf{od}\ [\!]\ \mathsf{true} \to h := h\ \mathsf{fi}, \tag{13}$$

which evidently does not reveal any information assuming an scheduler that makes non-deterministic 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 $\mathscr{I}_1$ 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 characterise these restrictions:

(iii) $\mathsf{Ind}[\mathscr{I}_1] \to \big(\ (\mathsf{EG}\,\neg\mathsf{mid}) \to \mathsf{AG}\,\neg\mathsf{end}\ \big)$
(iv) $\mathsf{Ind}[\mathscr{I}_1] \to \big(\ (\mathsf{EG}\,\neg\mathsf{mid}) \to \mathsf{AG}(\mathsf{mid} \to \mathsf{EG}\,\neg\mathsf{end})\ \big)$
(iv^s) $\mathsf{Ind}[\mathscr{I}_1] \to \big(\ (\mathsf{AF}\,\mathsf{mid}) \to \mathsf{AF}\,\mathsf{end}\ \big)$

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

10. Related work

Type-based analyses are by far the most common method used for enforcing information flow policies of programs – see, for example, Sabelfeld and Myers (2003). However, there is a growing body of work that pursues similar goals to ours, namely, enforcing non-interference using logical methods.

Work using self-composition

Our work was inspired by earlier results in Joshi and Leino (2000), which provide a characterisation of non-interference using weakest precondition calculi. Their characterisation, like ours, can be applied to a variety of programming constructs, including non-deterministic constructs, and can handle termination-sensitive non-interference. Their use of cylinders means they do not need to resort to self-composition; on the other hand, their approach is limited to weakest precondition calculi.

The idea of self-composition also appears in Darvas *et al.* (2005), which suggests that dynamic logic can be used to verify non-interference policies (termination-sensitive and

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

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; also, they do not discuss completeness issues.

The idea of self-composition has been explored further in a series of recent papers. For example, Terauchi and Aiken (2005) 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 the presence of certain predicates, for example, 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 sort of copy propagation optimisation for self-composed assignments with low expressions to variables. For example, if the program

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

has public variables x and z and confidential variable y, then 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 then show that non-interference is an instance of a 2-safety property. More recently, Clarkson and Schneider (2008) generalised this work to consider hyperproperties, which cover both liveness and safety, and generalise 2-safety to n-safety.

Moving closer to realistic programming languages, Jacobs and Warnier (2003) provides a method for verifying 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 is no precise analysis of the form of non-interference enforced by their method. More recently, Dufay *et al.* (2005) experimented with verification of information flow for Java programs using self-composition and JML specifications (Leavens *et al.* 1998); more precisely, they used the Krakatoa tool (Marché *et al.* 2004) to validate data mining algorithms. Their work is more oriented towards applications and does not formally justify self-composition. However, Naumann (2006) recently systematised and formally justified the modeling of information flow policies for Java programs using JML specifications. Naumann's work was heavily influenced by Benton (2004), which develops a relational Hoare logic for a simple imperative language.

In a concurrent setting, Huisman *et al.* (2006) recently proposed a characterisation of observational determinism (Zdancewic and Myers 2003) using self-composition. Their characterisation uses temporal logics and is thus amenable to model-checking after a suitable program abstraction has been constructed. On the negative side, Alur *et al.* (2006) established that a more general notion of confidentiality than non-interference cannot be characterised using self-composition.

More recently, a number of papers have explored self-composition in connection with quantitative analysis of information flow. For example, Backes *et al.* (2009) uses ideas of self-composition to discover automatically paths that yield information leaks,

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

and then uses this information to quantify the amount of information leakage. More recently, Yasuoka and Terauchi (2010) explored the possibility of expressing quantitative information flow policies as k-safety properties.

Dedicated logics and decision procedures

Andrews and Reitman (1980) was amongst the first papers to develop proof rules for reasoning about information flow for a concurrent imperative language. More recently, there have been several papers that use specific logics for enforcing non-interference.

Using the framework of abstract interpretation, Giacobazzi and Mastroeni (2004b) provides a proof method for proving abstract non-interference. This line of work has been extended more recently to Java bytecode in Zanardini (2006).

Using a dedicated logic based on the notion of independence, Amtoft *et al.* (2006) proposes 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 (Banerjee *et al.* 2007), and to conditional information flow (Amtoft and Banerjee 2007).

Dam (2006) provides a sound and complete proof procedure for verifying 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 of automating 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 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. An immediate consequence of the soundness of Volpano *et al.* (1996)'s type system and our characterisation of non-interference is 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 further explored the interplay between type systems and program logics and provided a systematic method for deriving logical proofs of non-interference from type derivations. In particular, Beringer and Hofmann have explored a semantical notion of self-composition, which dispenses with reasoning on a self-composed program, and shown 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.* (2007) encode the flow-sensitive type system of Hunt and Sands (2006) into an extension of dynamic logic with updates.

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

11. Conclusions

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 us to prove the soundness of type systems. One natural direction for further research is to provide similar characterisations for other notions of non-interference, and perhaps for other security properties such as anonymity.

Appendix A. Proof of Theorem 3

A.1. *Termination-sensitive case*

We have:

$$S \text{ is TS } (\mathscr{I}_1, \mathscr{I}_2)\text{-secure}$$

if and only if (by Corollary 1)

$$S \overset{\xi,\mathscr{I}_1}{\underset{\xi,\mathscr{I}_2}{\rhd\!\approx}} S[\xi]$$

if and only if (by Definition 2)

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

if and only if (by properties (a) and (b) of $\rhd$ (see Section 4) and Fact 1 (2) for $\Leftarrow$)

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

if and only if (by logic)

$$\left.\begin{array}{l}\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' : \\ \quad \mu_1 \oplus \mu_2 \sim_\xi^{\mathscr{I}_1} \mu_1 \oplus \mu_2 \\ \quad \Rightarrow \forall \mu_1' : \mathsf{var}(\mu_1') = \mathsf{var}(S) : \\ \qquad \neg(\ (S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2)\ ) \vee \\ \qquad \exists \mu_2' : \mathsf{var}(\mu_2') = \mathsf{var}(S)' : \\ \qquad\quad (S \rhd S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2') \ \wedge \ \mu_1' \oplus \mu_2' \sim_\xi^{\mathscr{I}_2} \mu_1' \oplus \mu_2'.\end{array}\right\} \quad (14)$$

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

Working in the opposite direction of the implication, we have

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

if and only if (by logic)

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

We now show that (14) and (15) are equivalent by 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 as follows:

(15) holds

if and only if (since $\neg((S, \mu_1 \oplus \mu_2)\bot)$ and by properties (a) and (b) of $\triangleright$ and the uniqueness of μ_1'')

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

if and only if (the first conjunct holds because of the uniqueness of μ_1'')

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

if and only if (by logic and determinism for $\Leftarrow$)

(14) holds.

We can adapt this proof so that it works just 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 $\mu_1', \mu_1'' \in \Theta$, we have $\mathsf{v}(\mu_1', x) = \mathsf{v}(\mu_1'', x)$ 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 $\mu_1', \mu_1'' \in \Theta$,

$$\mu_1' \oplus \mu_2' \sim_{\xi}^{\mathscr{I}_2} \mu_1' \oplus \mu_2' \quad \text{iff} \quad \mu_1'' \oplus \mu_2' \sim_{\xi}^{\mathscr{I}_2} \mu_1'' \oplus \mu_2' \quad (16)$$

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

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

(15) holds

if and only if (by $\neg((S,\mu_1\oplus\mu_2)\bot)$)

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

if and only if (by Property (b) of $\triangleright$ and Fact 1 (2))

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

if and only if (by $\Theta\neq\varnothing$, observation (16) and logic)

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

if and only if (the first conjunct holds by the definition of Θ)

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

if and only if (by logic and the definition of Θ for implication $\Leftarrow$)

(14) holds.

A.2. *Termination-insensitive case*

S is TI $(\mathscr{I}_1,\mathscr{I}_2)$-secure if and only if (by Corollary 1)

$$S\ \trianglerighteq_{\mathscr{I}_2}^{\mathscr{I}_1} S[\xi]$$

if and only if (by Definition 2)

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

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

if and only if (by properties (a) and (b) of $\triangleright$ and Fact 1 (2) for implication $\Leftarrow$)

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

if and only if (by Claim 1 below)

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

if and only if (by logic)

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

if and only if (by properties (a) and (b) of $\triangleright$ and Fact 1 (2))

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

if and only if (by logic)

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

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

if and only if (by determinism ($\mu_2' = \mu_2''$) and logic) (17)

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

Step (17) can also be justified by observational equivalence, in this case taking into account the fact that $\mathsf{v}(\mu_2', x) = \mathsf{v}(\mu_2'', x)$ for every $x \in \mathsf{var}(S)'$, and hence

$$\mu_1' \oplus \mu_2' \sim_\xi^{\mathscr{I}_2} \mu_1' \oplus \mu_2'$$

if and only if

$$\mu_1' \oplus \mu_2'' \sim_\xi^{\mathscr{I}_2} \mu_1' \oplus \mu_2''.$$

This proves the termination-insensitive case of Theorem 4.

Claim 1. Let μ_1, μ_2, μ_1' be 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

$$(S[\xi], \mu_1' \oplus \mu_2)\bot$$

if and only if

$$\forall \mu_2'' : \mathsf{var}(\mu_2'') = \mathsf{var}(S)' : \neg(\ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2'')\ ).$$

Proof. First note that $(S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2)$ because of property (b) of $\triangleright$ and Fact 1 (2). So we have:

$$(S[\xi], \mu_1' \oplus \mu_2)\bot$$

if and only if (by Fact 1 (1))

$$\neg\exists \mu_2'' : \mathsf{var}(\mu_2'') = \mathsf{var}(S)' : (S[\xi], \mu_1' \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2'')$$

if and only if (by the previous observation following the hypothesis of the claim and logic)

$$(S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2) \\ \Rightarrow \forall \mu_2'' : \mathsf{var}(\mu_2'') = \mathsf{var}(S)' : \neg(\ (S[\xi], \mu_1' \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2'')\ )$$

if and only if (by logic)

$$\forall \mu_2'' : \mathsf{var}(\mu_2'') = \mathsf{var}(S)' : \\ \neg(\ (S, \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2) \ \wedge \ (S[\xi], \mu_1' \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2'')\ )$$

if and only if (by property (b) of $\triangleright$ and Fact 1 (2) for $\Leftarrow$)

$$\forall \mu_2'' : \mathsf{var}(\mu_2'') = \mathsf{var}(S)' : \neg(\ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (\surd, \mu_1' \oplus \mu_2'')\ ).$$

□

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

Appendix B. Proof for Example 8

The invariant for the while do loop 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 into two parts and then, basically, focuses 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 into two salary lists, ps_{prev} and ps_{mis}. Now ps_{prev} contains the elements that have already been counted, while ps_{mis} contains those still to be counted. This (partial) counting 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 to the sum of all the salaries and n is the length of the original list, that is, the number of summed salaries.

For the verification of $\{\mathbf{I}_{sl}(\not\asymp)\}$ 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 have left as an exercise. We only omit a few proof obligations.

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

// *AV_SAL: the program itself as a first part*
// *of the composed program*
$p := lsalaries$
$s := 0$
$n := 0$

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

while $p \neq$ nil do
  $n := n + 1$
  $s_{aux} := p.salary$
  $s := s + s_{aux}$
  $p_{aux} := p.next$
  $p := p_{aux}$
od
$a_l := s/n$

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

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

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

$p' := lsalaries'$

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

$s' := 0$

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

$n' := 0$

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

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

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

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

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

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

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

$n' := n' + 1$

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

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

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

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

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

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

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

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

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

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

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

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

od

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

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

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

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

Appendix C. Proofs for characterisation of security in CTL – TS Case

We have:

$$S \text{ is TS } (\mathscr{I}_1, \mathscr{I}_2)\text{-secure}$$

if and only if (by Corollary 1)

$$S \overset{id,\mathscr{I}_1}{\underset{id,\mathscr{I}_2}{\rhd\!\approx}} S[\xi]$$

if and only if (by Definition 2)

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

if and only if (by the satisfaction of Ind, end and Fact 1 (1)

$$\begin{array}{l}\forall \mu_1, \mu_2, \mu'_1 : \mathsf{var}(\mu_1) = \mathsf{var}(\mu'_1) = \mathsf{var}(S) \wedge \mathsf{var}(\mu_2) = \mathsf{var}(S)' : \\ \quad \big(\ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \models \mathsf{Ind}[\mathscr{I}_1] \ \wedge \ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (S[\xi], \mu'_1 \oplus \mu_2)\ \big) \\ \quad \Rightarrow \exists c : (S[\xi], \mu'_1 \oplus \mu_2) \rightsquigarrow^* c \ \wedge \ c \models \mathsf{Ind}[\mathscr{I}_2] \wedge \mathsf{end}\end{array}$$

if and only if (by the semantics of EF and logic)

$$\begin{array}{l}\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' : \\ \quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \models \mathsf{Ind}[\mathscr{I}_1] \\ \quad \Rightarrow \forall \mu'_1 : \mathsf{var}(\mu'_1) = \mathsf{var}(S) : \\ \qquad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (S[\xi], \mu'_1 \oplus \mu_2) \Rightarrow (S[\xi], \mu'_1 \oplus \mu_2) \models \mathsf{EF}(\mathsf{Ind}[\mathscr{I}_2] \wedge \mathsf{end})\end{array}$$

if and only if (by logic)

$$\begin{array}{l}\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' : \\ \quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \models \mathsf{Ind}[\mathscr{I}_1] \\ \quad \Rightarrow \forall \mu : (\ \exists \mu'_1 : \mathsf{var}(\mu'_1) = \mathsf{var}(S) : \mu = \mu'_1 \oplus \mu_2 \wedge (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (S[\xi], \mu'_1 \oplus \mu_2)\ ) \\ \qquad \Rightarrow (S[\xi], \mu) \models \mathsf{EF}(\mathsf{Ind}[\mathscr{I}_2] \wedge \mathsf{end})\end{array}$$

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

if and only if (by property (a) of $\triangleright$)

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

if and only if (by logic and Fact 1 (1))

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

if and only if (by property (c) of $\triangleright$ for $\Rightarrow$ and property (a) of $\triangleright$ for $\Leftarrow$)

$$\begin{array}{l}\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' : \\ \quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \models \mathsf{Ind}[\mathscr{I}_1] \\ \quad \Rightarrow \forall \mu : (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* (S[\xi], \mu) \ \Rightarrow \ (S[\xi], \mu) \models \mathsf{EF}(\mathsf{Ind}[\mathscr{I}_2] \wedge \mathsf{end})\end{array}$$

if and only if (by logic and the satisfaction of mid)

$$\begin{array}{l}\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' : \\ \quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \models \mathsf{Ind}[\mathscr{I}_1] \\ \quad \Rightarrow \forall c : \big(\ (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* c \ \wedge \ c \models \mathsf{mid} \ \big) \ \Rightarrow \ c \models \mathsf{EF}(\mathsf{Ind}[\mathscr{I}_2] \wedge \mathsf{end})\end{array}$$

if and only if (by the semantics of $\rightarrow$, and logic)

$$\begin{array}{l}\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' : \\ \quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \models \mathsf{Ind}[\mathscr{I}_1] \\ \quad \Rightarrow \forall c : (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \rightsquigarrow^* c \Rightarrow c \models (\ \mathsf{mid} \rightarrow \mathsf{EF}(\mathsf{Ind}[\mathscr{I}_2] \wedge \mathsf{end}) \ )\end{array}$$

if and only if (by the semantics of AG and $\rightarrow$)

$$\begin{array}{l}\forall \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \ \wedge \ \mathsf{var}(\mu_2) = \mathsf{var}(S)' : \\ \quad (S \triangleright S[\xi], \mu_1 \oplus \mu_2) \ \models \ \mathsf{Ind}[\mathscr{I}_1] \rightarrow \mathsf{AG}(\ \mathsf{mid} \rightarrow \mathsf{EF}(\mathsf{Ind}[\mathscr{I}_2] \wedge \mathsf{end}) \ )\end{array}$$

if and only if (since $S \models \Phi$ if and only if $\forall \mu : (S, \mu) \models \Phi$ by definition, and, since here memory operations are functions, $\forall \mu : \exists \mu_1, \mu_2 : \mathsf{var}(\mu_1) = \mathsf{var}(S) \wedge \mathsf{var}(\mu_2) = \mathsf{var}(S)' \wedge \mu = \mu_1 \oplus \mu_2$)

$$S \triangleright S[\xi] \ \models \ \mathsf{Ind}[\mathscr{I}_1] \rightarrow \mathsf{AG}(\ \mathsf{mid} \rightarrow \mathsf{EF}(\mathsf{Ind}[\mathscr{I}_2] \wedge \mathsf{end}) \ ).$$

Acknowledgments

We would like to thank an anonymous referee for his/her thorough work in reviewing our submission, which enabled us to make significant improvements in the quality of this paper.

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

References

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

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

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

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

Backes, M., Köpf, B. and Rybalchenko, A. (2009) Automatic Discovery and Quantification of Information Leaks. In: *Proceedings 30th IEEE Symposium on Security and Privacy (S&P '09)*, IEEE Computer Society Press 141–153.

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

Barthe, G., D'Argenio, P. and Rezk, T. (2004) Secure Information Flow by Self-Composition. In: Foccardi, R. (ed.) *Proceedings of CSFW'04*, IEEE Computer Society Press 100–114.

Benton, N. (2004) Simple relational correctness proofs for static analyses and program transformations. In: *Proceedings of POPL'04*, ACM Press. 14–25.

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

Clarkson, M. R. and Schneider, F. B. (2008) Hyperproperties. In: *Computer Security Foundations Symposium*, IEEE Computer Society Press 51–65.

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

Cohen, E. S. (1978) Information transmission in sequential programs. In: DeMillo, R. A., Dobkin, D. P., Jones, A. K. and Lipton, R. J. (eds.) *Foundations of Secure Computation*, Academic Press 297–335.

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

Dam, M. (2006) Decidability and proof systems for language-based noninterference relations. In: *Proceedings of POPL'06*, ACM Press 67–78.

Darvas, A., Hähnle, R. and Sands, D. (2005) A theorem proving approach to analysis of secure information flow. In: Hutter, D. and Ullmann, M. (eds.) Security in Pervasive Computing. *Springer-Verlag Lecture Notes in Computer Science* **3450** 193–209. (Preliminary version in the informal proceedings of WITS'03.)

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

Dijkstra, E. (1997) *A Discipline of Programming*, Prentice Hall.

Dufay, G., Felty, A. P. and Matwin, S. (2005) Privacy-sensitive information flow with JML. In: Nieuwenhuis, R. (ed.) Proceedings of CADE'05. *Springer-Verlag Lecture Notes in Computer Science* **3632** 116–130.

Giacobazzi, R. and Mastroeni, I. (2004a) Abstract non-interference: Parameterizing non-interference by abstract interpretation. In: *Proceedings of the* 31^{th} *ACM Symposium on Principles of Programming Languages*, ACM Press 186–197.

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

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

Goguen, J. and Meseguer, J. (1982) Security policies and security models. In: *Proceedings of SOSP'82*, IEEE Computer Society Press 11–22.

Hähnle, R., Pan, J., Rümmer, P. and Walter, D. (2007) Integration of a security type system into a program logic. In: Proceedings 2nd Symposium on Trustworthy Global Computing. *Springer-Verlag Lecture Notes in Computer Science* **4661** 116–131.

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

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

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

Ishtiaq, S. and O'Hearn, P. (2001) Bi as an assertion language for mutable data structures. In: *Proceedings of the* 28^{th} *ACM Symposium on Principles of Programming Languages*, ACM Press 14–26.

Jacobs, B. and Warnier, M. (2003) Formal proofs of confidentiality in java programs (manuscript).

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

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

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

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

Mclean, J. (1994) A general theory of composition for trace sets closed under selective interleaving functions. In: *In Proceedings IEEE Symposium on Security and Privacy*, IEEE Computer Society Press 79–93.

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

Necula, G. (1997) Proof-carrying code. In: *Proceedings of the* 25^{th} *ACM Symposium on Principles of Programming Languages*, ACM Press 106–119.

Necula, G. (1998) *Compiling with Proofs*, Ph.D. thesis, Carnegie Mellon University. (Available as Technical Report CMU-CS-98-154.)

Pottier, F. (2002) A simple view of type-secure information flow in the pi-calculus. In: *CSFW*, IEEE Computer Society Press 320–330.

Reynolds, J. (2000) Intuitionistic reasoning about shared mutable data structure. In: Davies, J., Roscoe, B. and Woodcock, J. (eds.) *Millennial Perspectives in Computer Science*, Palgrave 303–321.

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

Sabelfeld, A. and Myers, A. (2003) Language-based information-flow security. *IEEE Journal on Selected Areas in Communication* **21** 5–19.

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press

Sabelfeld, A. and Myers, A. C. (2004) A model for delimited information release. In: Futatsugi, K., Mizoguchi, F. and Yonezaki, N. (eds.) Software Security – Theories and Systems: Proceedings International Symposium on Software Security (ISSS'03). *Springer-Verlag Lecture Notes in Computer Science* **3233** 174–191.

Sabelfeld, A. and Sands, D. (2005) Dimensions and principles of declassification. In: *CSFW*, 255–269.

Smith, G. and Volpano, D. (1998) Secure Information Flow in a Multi-threaded Imperative Language. In: *Proceedings of the* 25^{th} *ACM Symposium on Principles of Programming Languages*, ACM Press 355–364.

Terauchi, T. and Aiken, A. (2005) Secure information flow as a safety problem. In: Hankin, C. and Siveroni, I. (eds.) Static Analysis Symposium. *Springer-Verlag Lecture Notes in Computer Science* **3672** 352–367.

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

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

Yasuoka, H. and Terauchi, T. (2010) Quantitative information flow – verification hardness and possibilities. In: *Proceedings CSF'10*, IEEE Computer Society Press.

Zanardini, D. (2006) Abstract Non-Interference in a fragment of Java bytecode. In: *Proceedings of the ACM Symposium on Applied Computing (SAC)*, ACM Press 1822–1826.

Zdancewic, S. and Myers, A. (2003) Observational determinism for concurrent program security. In: *Proceedings of CSFW'03*, IEEE Computer Society Press 29–43.

https://doi.org/10.1017/S0960129511000193 Published online by Cambridge University Press