Bridging Language-Based and Process Calculi Security*

Riccardo Focardi[1], Sabina Rossi[1], and Andrei Sabelfeld[2]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia,
30172 Venezia, Italy
{focardi, srossi}@dsi.unive.it
[2] Dept. of Computer Science, Chalmers University of Technology,
41296 Göteborg, Sweden
andrei@cs.chalmers.se

Abstract. Language-based and process calculi-based information security are well developed fields of computer security. Although these fields have much in common, it is somewhat surprising that the literature lacks a comprehensive account of a formal link between the two disciplines. This paper develops such a link between a language-based specification of security and a process-algebraic framework for security properties. Encoding imperative programs into a CCS-like process calculus, we show that timing-sensitive security for these programs exactly corresponds to the well understood process-algebraic security property of persistent bisimulation-based nondeducibility on compositions (P_BNDC). This rigorous connection opens up possibilities for cross-fertilization, leading to both flexible policies when specifying the security of heterogeneous systems and to a synergy of techniques for enforcing security specifications.

1 Introduction

As computing systems are becoming increasingly complex, security challenges become increasingly versatile. In the presence of such challenges, we believe that practical security solutions are unlikely to emerge from a single theoretical framework, but rather need to be based on a combination of different specialized approaches. The goal of this paper is to develop a flexible way of specifying the security of heterogeneous systems—using a combination of language-based definitions and process-algebraic ones. The intention is to be able to specify security partly by language-based security models (e.g., for parts of the system that are implemented by code with no communication) and partly by process-algebraic models (e.g., communication-intensive parts of the system). This combined approach empowers us with a synergy of techniques for enforcing security properties (e.g., combining security type systems with process equivalence checking) to analyze parts of the system separately and yet establish the security of the entire system.

Language-based information security [27] and process calculus-based information security [7, 25] are well developed fields of computer security. Although process calculi are programming languages, there are different motivations and traditions in addressing information security by the two communities. While the former is concerned with

* This work was supported by the EU-FET project MyThS (IST-2001-32617).

V. Sassone (Ed.): FOSSACS 2005, LNCS 3441, pp. 299–315, 2005.

preventing secret data from being leaked through the execution of programs, the latter deals with preventing secret events from being revealed through the execution of communicating processes. Although these fields have much in common (e.g., both rely on *noninterference* [12] as a baseline security policy stating that secrets do not interfere with the attacker-observable behavior of the system), it is somewhat surprising that the literature lacks a comprehensive account of a formal link between the two disciplines (which in particular means that it has not been established whether the interpretations of noninterference by the two disciplines are compatible).

This paper develops a rigorous link between a language-based specification of security for imperative programs and a process-algebraic framework of security properties. More specifically, we link two compositional security properties: a timing-sensitive security characterization for a simple imperative language and a persistent security characterization for a CCS-like process calculus. We achieve this connection through the following steps: (i) we uniform the semantics of the imperative language to the standard *Labelled Transition System* semantics of process calculi, by making read/write memory actions explicitly observable as labelled transitions; (ii) based on this semantics, we formalize low level observations in the imperative language in terms of a bisimulation relation; (iii) we encode the programming language into the process calculus, ensuring a lock-step semantic relation between the source and target languages; we prove that the new bisimulation notion for the imperative language is preserved by the encoding; (iv) this tight relation reveals some unexpected uniformities allowing us to precisely identify what the program security characterization corresponds to in the process-calculus world: it turns out to be the well understood property of persistent bisimulation-based nondeducibility on compositions (or P_BNDC).

Such a link opens up various possibilities for cross-fertilization, leading to flexible policies when specifying the security of complex systems and to a rich combination of techniques for enforcing security specifications. Finding exactly what property from the family of process-algebraic properties [7, 9] corresponds to the language-based timing-sensitive security sheds valuable light on the nature of the language-based property. As a direct benefit, the results of this paper enable us to use security checkers based on process-equivalence checking (such as CoSeC [6] and CoPS [23], with the latter one based precisely on P_BNDC) for certifying language-based security.

For clarity, this paper uses a simple sequential language. However, it is a distributed setting that will enable us to fully capitalize on the formal connection. Indeed, the security specifications for both the source (imperative) and target (process algebraic) languages are compositional [28, 9]. Because the source-language security specification is suitable for both multithreaded [28] and distributed [26, 21] settings, we are confident that the formal link established in this paper can be generalized to a distributed scenario, where different components can be analyzed with specialized techniques. For example, communication-intensive parts of the system (where conservative language-based security mechanisms for the source language such as type systems are too restrictive) can be analyzed at the level of the target language, gaining on the precision of the analysis.

The paper is organized as follows. Section 2 presents the source imperative language Imp and the target process-algebraic language VSPA. Section 3 develops an encoding of the source language into the target language and demonstrates a semantic relation

between Imp's programs and their VSPA's translations. Section 4 establishes a formal connection between the security properties of the two languages. The paper closes by discussing related work in Section 5 and conclusions and future work in Section 6.

The proofs of the results presented in this paper are reported in [10].

2 The Source Language and Target Calculus

In this section, we present Imp, the source imperative language, and VSPA, the target process calculus, along with security definitions for the respective languages.

2.1 The Imp Programming Language

We consider a simple sequential programming language, Imp [30], described by the following grammar:

$$B, Exp ::= F(Id, \ldots, Id)$$
$$C ::= \mathsf{stop} \mid \mathsf{skip} \mid Id := Exp \mid C; C \mid \mathsf{if}\ B\ \mathsf{then}\ C\ \mathsf{else}\ C \mid \mathsf{while}\ B\ \mathsf{do}\ C$$

Let $C, D, \ldots$ range over commands (programs), $Id, Id_1, \ldots$ range over identifiers (variables), $B, B_1, \ldots, Exp, Exp_1, \ldots$ range over boolean and arithmetic expressions, respectively, $F, F_1, \ldots$ range over function symbols, and, finally, $v, v_1, \ldots$ range over the set of basic values Val. For simplicity, but without loss of generality, we assume that exactly one function symbol occurs in an expression.

A *configuration* is a pair $\langle C, s\rangle$ of a command C and a state (memory) s. A *state* s is a finite mapping from variables to values. The small-step semantics are given by transitions

$$\frac{}{\langle \mathsf{skip}, s\rangle \overset{\mathtt{tick}}{\rightarrow} \langle \mathsf{stop}, s\rangle}$$

$$\frac{\langle Exp, s\rangle \downarrow v}{\langle Id := Exp, s\rangle \overset{\mathtt{tick}}{\rightarrow} \langle \mathsf{stop}, [Id \mapsto v]s\rangle}$$

$$\frac{\langle C_1, s\rangle \overset{\mathtt{tick}}{\rightarrow} \langle \mathsf{stop}, s'\rangle}{\langle C_1; C_2, s\rangle \overset{\mathtt{tick}}{\rightarrow} \langle C_2, s'\rangle} \qquad \frac{\langle C_1, s\rangle \overset{\mathtt{tick}}{\rightarrow} \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \overset{\mathtt{tick}}{\rightarrow} \langle C_1'; C_2, s'\rangle}$$

$$\frac{\langle B, s\rangle \downarrow \mathsf{True}}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s\rangle \overset{\mathtt{tick}}{\rightarrow} \langle C_1, s\rangle}$$

$$\frac{\langle B, s\rangle \downarrow \mathsf{False}}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s\rangle \overset{\mathtt{tick}}{\rightarrow} \langle C_2, s\rangle}$$

$$\frac{\langle B, s\rangle \downarrow \mathsf{True}}{\langle \mathsf{while}\ B\ \mathsf{do}\ C, s\rangle \overset{\mathtt{tick}}{\rightarrow} \langle C; \mathsf{while}\ B\ \mathsf{do}\ C, s\rangle}$$

$$\frac{\langle B, s\rangle \downarrow \mathsf{False}}{\langle \mathsf{while}\ B\ \mathsf{do}\ C, s\rangle \overset{\mathtt{tick}}{\rightarrow} \langle \mathsf{stop}, s\rangle}$$

Fig. 1. Small-step semantics of Imp commands

between configurations, defined by standard transition rules (see Fig. 1). Arithmetic and boolean expressions are executed atomically by $\downarrow$ transitions. The $\stackrel{\texttt{tick}}{\rightarrow}$ transitions are deterministic. The general form of a deterministic transition is $\langle C, s\rangle \stackrel{\texttt{tick}}{\rightarrow} \langle C', s'\rangle$. Here, one step of computation starting with a command C in a state s gives a new command C' and a new state s'. There are no transitions from configurations that contain the terminal program stop. We write $[Id \mapsto v]s$ for the state obtained from s by setting the image of Id to v. For example, the assignment rule describes one step of computation that leads to termination with the state updated according to the value of the expression on the right-hand side of the assignment.

Security Specification. We assume that the set of variables is partitioned into *high* and *low* security classes corresponding to high and low confidentiality levels. Note that our results are not specific to this security structure (which is adopted for simplicity)—a generalization to an arbitrary security lattice is straightforward. Variables h and l will denote typical high and low variables respectively. Two states s and t are *low-equal* $s =_L t$ if the low components of s and t are the same.

Confidentiality is preserved by a computing system if low-level observations reveal nothing about high-level data. The notion of noninterference [12] is widely used for expressing such confidentiality policies. Intuitively, noninterference means that low-observable behavior is unchanged as high inputs are varied. The indistinguishability of behavior for the attacker can be represented naturally by the notion of *bisimulation* (e.g., [7, 28]). The following definition is recalled from [28]:

Definition 1. Strong low-bisimulation $\approxeq_L$ *is the union of all symmetric relations* R *such that if* $C\ R\ D$ *then for all states* s *and* t *such that* $s =_L t$ *whenever* $\langle C, s\rangle \stackrel{\texttt{tick}}{\rightarrow} \langle C', s'\rangle$ *then there exist* D' *and* t' *such that* $\langle D, t\rangle \stackrel{\texttt{tick}}{\rightarrow} \langle D', t'\rangle$, $s' =_L t'$, *and* $C'\ R\ D'$.

Intuitively, two programs C and D are strongly low-bisimilar if varying the high parts of memories at any point of computation does not introduce any difference between the low parts of the memories throughout the computation. Protecting variations at any point of computation results in a rather restrictive security condition. However, this restrictiveness is justified in a concurrent setting (which is the ultimate motivation of our work) when threads may introduce secrets into high memory at any computation step. Based on this notion of low-bisimulation, a definition of security is given in [28]:

Definition 2. *A program* C *is secure if and only if* $C \approxeq_L C$.

Examples. Because the underlying low-bisimulation is strong, or lock-step, it captures timing-sensitive security of programs. Below, we exemplify different kinds of information flow handled by the security definition:

$l := h$ This is an example of an *explicit flow*. To see that this program is insecure according to Definition 2, take some s and t that are the same except $s(h) = 0$ and $t(h) = 1$. Since $\langle l := h, s\rangle \stackrel{\texttt{tick}}{\rightarrow} \langle \mathsf{stop}, [l \mapsto 0]s\rangle$ and $\langle l := h, t\rangle \stackrel{\texttt{tick}}{\rightarrow} \langle \mathsf{stop}, [l \mapsto 1]t\rangle$ hold, the resulting memories are not low-equal. Because these are the only possible transitions for both configurations, we have $l := h \not\approxeq_L l := h$.

if $h > 0$ **then** $l := 1$ **else** $l := 0$ This exemplifies an *implicit flow* [4] through branching on a high condition. If the computation starts with low-equal memories s and t that are the same except $s(h) = 0$ and $t(h) = 1$, then, after one step of computation (the test of the condition), the memories are still low-equal. However, after another computation step they become different in l (0 or 1, depending on the initial value of h). Because these are the only possible transitions for configurations with both s and t, the program is not self-low-similar and thus is insecure.

while $h > 0$ **do** $h := h - 1$ Assuming the worst-case scenario, an attacker may observe the timing of program execution. The attacker may learn the value of h from the timing behavior of the program above. This is an instance of a *timing covert channel* [19]. The program is rightfully rejected by Definition 2. Indeed, take some s and t that are the same except $s(h) = 1$ and $t(h) = 0$. We have $\langle \mathsf{while}\ h > 0\ \mathsf{do}\ h := h - 1, s\rangle \stackrel{\mathtt{tick}}{\rightarrow} \langle h := h - 1; \mathsf{while}\ h > 0\ \mathsf{do}\ h := h - 1, s\rangle \stackrel{\mathtt{tick}}{\rightarrow} \langle \mathsf{while}\ h > 0\ \mathsf{do}\ h := h - 1, [h \mapsto 0]s\rangle \stackrel{\mathtt{tick}}{\rightarrow} \langle \mathsf{stop}, [h \mapsto 0]s\rangle$ but $\langle \mathsf{while}\ h > 0\ \mathsf{do}\ h := h - 1, t\rangle \stackrel{\mathtt{tick}}{\rightarrow} \langle \mathsf{stop}, t\rangle \stackrel{\mathtt{tick}}{\not\rightarrow}$ with no transition from the latter configuration to match the transitions of the previous sequence.

The examples above are insecure. Here is an instance of a secure program:

if $h = 1$ **then** $h := h + 1$ **else skip** Indeed, neither the low part of the memory nor the timing behavior depends on the value of h. A suitable symmetric relation that makes this program low-bisimilar to itself is, e.g., the relation $\{(\mathsf{if}\ h = 1\ \mathsf{then}\ h := h + 1\ \mathsf{else\ skip}, \mathsf{if}\ h = 1\ \mathsf{then}\ h := h + 1\ \mathsf{else\ skip}), (h := h + 1, \mathsf{skip}), (\mathsf{skip}, h := h + 1), (h := h + 1, h := h + 1), (\mathsf{skip}, \mathsf{skip}), (\mathsf{stop}, \mathsf{stop})\}$.

2.2 The VSPA Calculus

The *Value-passing Security Process Algebra* (VSPA, for short) is a variation of Milner's value-passing CCS [22], where the set of visible actions is partitioned into high-level actions and low-level ones in order to specify multilevel-security systems.

Let $E, E_1, E_2, \ldots$ range over *processes*, $x, x_1, x_2, \ldots$ range over *variables*, c, c_1, $c_2, \ldots$ range over *input channels*, and $\bar{c}$, $\bar{c}_1$, $\bar{c}_2, \ldots$ range over *output channels*. As for Imp, let $B, B_1, \ldots, Exp, Exp_1, \ldots$ range over boolean and arithmetic expressions, respectively, $F, F_1, \ldots$ range over function symbols, and, finally, $v, v_1, \ldots$ range over the set of basic values Val. (The set of basic values Val, and boolean/arithmetic expressions are the same as in Imp.) The set of visible actions is $\mathcal{L} = \{c(v) \mid v \in Val\} \cup \{\bar{c}(v) \mid v \in Val\}$ where $c(v)$ and $\bar{c}(v)$ represent the input and the output of value v over the channel c, respectively. The syntax of VSPA processes is defined as follows:

$$\begin{aligned} E &::= \mathbf{0} \mid c(x).E \mid \bar{c}(Exp).E \mid \tau.E \mid E_1 + E_2 \mid E_1|E_2 \mid E \setminus R \mid E[g] \mid \\ &\qquad A(Exp_1, \ldots, Exp_n) \mid \mathbf{if}\ B\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2 \\ B, Exp &::= F(x_1, \ldots, x_n) \end{aligned}$$

Each constant A is associated with a definition $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} E$, where $x_1, \ldots, x_n$ are distinct variables and E is a VSPA process whose only free variables are $x_1, \ldots, x_n$. R is a set of channels and g is a function relabeling channel names which preserves the

$$c(x).E \xrightarrow{c(v)} E[v/x] \qquad \overline{c}(v).E \xrightarrow{\overline{c}(v)} E \qquad \tau.E \xrightarrow{\tau} E$$

$$\frac{E_1 \xrightarrow{a} E_1'}{E_1 + E_2 \xrightarrow{a} E_1'} \qquad \frac{E_2 \xrightarrow{a} E_2'}{E_1 + E_2 \xrightarrow{a} E_2'}$$

$$\frac{E_1 \xrightarrow{a} E_1'}{E_1|E_2 \xrightarrow{a} E_1'|E_2} \qquad \frac{E_2 \xrightarrow{a} E_2'}{E_1|E_2 \xrightarrow{a} E_1|E_2'} \qquad \frac{E_1 \xrightarrow{c(v)} E_1' \quad E_2 \xrightarrow{\overline{c}(v)} E_2'}{E_1|E_2 \xrightarrow{\tau} E_1'|E_2'}$$

$$\frac{E \xrightarrow{a} E'}{E[g] \xrightarrow{g(a)} E'[g]} \qquad \frac{E \xrightarrow{a} E' \quad a \notin R}{E \setminus R \xrightarrow{a} E' \setminus R}$$

$$\frac{E[v_1/x_1, \ldots, v_n/x_n] \xrightarrow{a} E' \quad A(x_1, \ldots, x_n) \stackrel{def}{=} E}{A(v_1, \ldots, v_n) \xrightarrow{a} E'}$$

$$\frac{E_1 \xrightarrow{a} E_1'}{\textbf{if } \mathsf{True} \textbf{ then } E_1 \textbf{ else } E_2 \xrightarrow{a} E_1'} \qquad \frac{E_2 \xrightarrow{a} E_2'}{\textbf{if } \mathsf{False} \textbf{ then } E_1 \textbf{ else } E_2 \xrightarrow{a} E_2'}$$

Fig. 2. VSPA operational semantics

complementation operator $\bar{\cdot}$. Finally, the set of channels is partitioned into *high-level* channels H and *low-level* ones L. By an abuse of notation, we write $c(v), \overline{c}(v) \in H$ whenever $c, \overline{c} \in H$, and similarly for L.

Intuitively, $\mathbf{0}$ is the empty process; $c(x).E$ is a process that reads a value $v \in Val$ from channel c assigning it to variable x; $\overline{c}(Exp).E$ is a process that evaluates expression Exp and sends the resulting value as output over c; $E_1 + E_2$ represents the nondeterministic choice between the two processes E_1 and E_2; $E_1|E_2$ is the parallel composition of E_1 and E_2, where executions are interleaved, possibly synchronized on complementary input/output actions, producing an internal action τ; $E \setminus R$ is a process E prevented from using channels in R; $E[g]$ is the process E whose channels are renamed *via* the relabeling function g; $A(Exp_1, ..., Exp_n)$ behaves like the respective definition where the variables $x_1, \cdots, x_n$ are substituted with the results of expressions $Exp_1, \cdots, Exp_n$; finally, **if** B **then** E_1 **else** E_2 behaves as E_1 if B evaluates to True and as E_2, otherwise. We implicitly equate processes whose expressions are substituted by the corresponding values, e.g., $\overline{c}(F(v_1, \ldots v_n)).E$ is the same as $\overline{c}(v).E$ if $F(v_1, \ldots v_n) = v$. This corresponds to the $\downarrow$ expression evaluation of Imp. The operational semantics of VSPA is given in Fig. 2. We denote by $\mathcal{E}$ the set of all VSPA processes and by $\mathcal{E}_H$ the set of all high-level processes, i.e., using only channels in H.

The *weak bisimulation* relation [22] equates two processes if they are able to mutually simulate each other step by step. Weak bisimulation does not care about internal τ actions. We write $E \stackrel{a}{\Longrightarrow} E'$ if $E(\xrightarrow{\tau})^* \xrightarrow{a} (\xrightarrow{\tau})^* E'$. Moreover, we let $E \stackrel{\hat{a}}{\Longrightarrow} E'$ stand for $E \stackrel{a}{\Longrightarrow} E'$ in case $a \neq \tau$, and for $E(\xrightarrow{\tau})^* E'$ in case $a = \tau$.

Definition 3 (Weak bisimulation). *A symmetric binary relation $R \subseteq \mathcal{E} \times \mathcal{E}$ over processes is a* weak bisimulation *if whenever $(E, F) \in R$ and $E \xrightarrow{a} E'$, then there exists F' such that $F \stackrel{\hat{a}}{\Longrightarrow} F'$ and $(E', F') \in R$.*

Two processes $E, F \in \mathcal{E}$ are *weakly bisimilar*, denoted by $E \approx F$, if there exists a weak bisimulation R containing the pair (E, F). The relation $\approx$ is the largest weak bisimulation and it is an equivalence relation [22].

Persistent BNDC Security. In [9] we give a notion of security for VSPA processes called *Persistent BNDC*, where *BNDC* stands for *Bisimulation-based Nondeducibility on Compositions*. *BNDC* [5] is a generalization to concurrent processes of noninterference [12], consisting of checking a process E against all high-level processes Π.

Definition 4 (BNDC). *Let* $E \in \mathcal{E}$. $E \in BNDC$ *iff* $\forall\ \Pi \in \mathcal{E}_H,\ E \setminus H \approx (E|\Pi) \setminus H$.

Intuitively, BNDC requires that high-level processes Π have no effect at all on the (low-level) execution of E.

To introduce *Persistent BNDC* (*P_BNDC*) we define a new observation equivalence where high-level actions *may* be ignored, i.e., they may be matched by zero or more τ actions. An action a is high if a is either an input $c(v)$ or an output $\overline{c}(v)$, over a high-level channel $c \in H$. Otherwise, a is low. We write $\tilde{a}$ to denote $\hat{a}$ if a is low, and a or $\hat{\tau}$ if a is high. We now define weak bisimulation up to high, by just using $\tilde{a}$ in place of $\hat{a}$, thus allowing high-level actions to be simulated by (possibly empty) sequences of τ's.

Definition 5 (Weak bisimulation up to high). *A symmetric binary relation* $R \subseteq \mathcal{E} \times \mathcal{E}$ *over processes is a* weak bisimulation up to high *if whenever* $(E, F) \in R$ *and* $E \xrightarrow{a} E'$, *then there exists* F' *such that* $F \stackrel{\tilde{a}}{\Longrightarrow} F'$ *and* $(E', F') \in R$.

We say that two processes E, F are *weakly bisimilar up to high*, written $E \approx_{\setminus H} F$, if $(E, F) \in R$ for some weak bisimulation up to high R.

Definition 6 (P_BNDC). *Let* $E \in \mathcal{E}$. $E \in P\_BNDC$ *iff* $E \setminus H \approx_{\setminus H} E$.

Intuitively, *P_BNDC* requires that forbidding any high-level activity (by restriction) is equivalent to ignoring it. For example, process $E \stackrel{\text{def}}{=} h.\bar{l} + \bar{l}$ is *P_BNDC* since the high level input h is simulated, in $E \setminus H$, by not moving. Indeed, the high level activity is not visible to the low level users who can only observe the low level output $\bar{l}$. Notice that this secure process allows some low level actions to follow high actions.

It has been proved [9] that *P_BNDC* corresponds to requiring *BNDC* over all the possible reachable states. This is why we call it *Persistent BNDC*.

Proposition 1. $E \in P\_BNDC$ *iff* $\forall\ E'$ *reachable from* $E, E' \in BNDC$.

Note that P_BNDC is similarly spirited to Imp's security definition. In particular, the Π process in $BNDC$ corresponds to the possibility for arbitrary changes in the high part of state over the computation. Further, persistence in P_BNDC corresponds to requiring strong low-bisimulation on reachable Imp commands. There are also obvious differences, highlighting the specifics of the application domains of the two security specifications: P_BNDC is concerned with protecting the occurrence of high events whereas program security protects high memories.

P_BNDC satisfies useful compositionality properties and is much easier to check than *BNDC*, since no quantification over all possible high-level processes is required.

$$\frac{s(Id) = v}{\langle C, s\rangle \overset{\overline{\texttt{eget}}_{Id}(v)}{\longrightarrow} \langle C, s\rangle} \qquad \langle C, s\rangle \overset{\texttt{eput}_{Id}(v)}{\longrightarrow} \langle C, [Id \mapsto v]s\rangle \qquad \frac{\langle C, s\rangle \overset{a}{\rightarrow} \langle C', s'\rangle \quad a \notin R}{\langle C, s\rangle \setminus R \overset{a}{\rightarrow} \langle C', s'\rangle \setminus R}$$

Fig. 3. Semantic rules for environment

Example. We give a very simple example of an insecure process. In particular, we show an indirect flow due to the possibility for a high-level user to lock and unlock a process:

$$E \overset{\text{def}}{=} \texttt{hlock.hunlock.}\overline{\texttt{l}} + \overline{\texttt{l}}$$

where `hlock` and `hunlock` are high-level channels and `l` is a low-level one. (To simplify we are not even sending values over channels.) At a first glance, process E seems to be secure as it always performs $\overline{\texttt{l}}$ before terminating, thus low-level users should deduce nothing of what is done at the high level. However, a high-level user might lock the process through `hlock` and never unlock it, thus leading to an unexpected behavior since $\overline{\texttt{l}}$ would be locked too. This ability for a high-level user to synchronize with a low-level one constitutes an indirect information flow and is detected by *P_BNDC* since $E \overset{\texttt{hlock}}{\longrightarrow} \texttt{hunlock.}\overline{\texttt{l}}$ cannot be simulated by $E \setminus H$. In fact, $E \setminus H$ can execute neither high-level actions nor τ ones, thus the only possibility it has to simulate `hlock` is not moving. However, this simulation is fine as long as the reached states are bisimilar up to high, i.e., $\texttt{hunlock.}\overline{\texttt{l}} \approx_{\setminus H} E \setminus H$, but this is not true.

3 Mapping Imp into VSPA

With the source and target languages in place, this section develops an encoding of the former into the latter. The encoding is done in two steps: enriching Imp's semantics with process calculi-style environment interaction rules and encoding the extended version of Imp into VSPA. A lock-step relation of Imp's executions with their VSPA's translations guarantees that the encoding is semantically adequate.

3.1 Extending Imp Semantics

The original definition of strong low-bisimulation (Definition 1) implicitly takes into account an environment that is capable of both reading from and writing to the state at any point of computation. Alternatively, and rather naturally, we can represent this environment explicitly, by the semantic rules for reading and modifying the state, depicted in Fig. 3. Reading the value v of a variable Id is observable by an action $\overline{\texttt{eget}}_{Id}(v)$; writing the value v to Id is observable by an action $\texttt{eput}_{Id}(v)$. (We adopt the process calculi convention of using $\bar{\cdot}$ to denote output actions.)

Assume $a \in \{\texttt{tick}, \overline{\texttt{eget}}_{\cdot}(\cdot), \texttt{eput}_{\cdot}(\cdot)\}$. Action a is *high* ($a \in H$) if for some high variable Id we have either $a = \overline{\texttt{eget}}_{Id}(\cdot)$ or $a = \texttt{eput}_{Id}(\cdot)$. Otherwise, a is *low* ($a \in L$). High and low actions represent high and low environments, respectively. Similarly to VSPA's restriction, we define a restriction on actions in the semantics for

Imp, also shown in Fig. 3. For a set of actions R, an R-restricted configuration $\langle C, s\rangle \setminus R$ behaves as $\langle C, s\rangle$ except that its communication on actions from R is prohibited. The restriction is helpful for relating the extended semantics to Imp's original semantics: configuration $\langle C, s\rangle \setminus \{\overline{\texttt{eget}}_{Id}(v), \texttt{eput}_{Id}(v) \mid Id \text{ is a variable and } v \in Val\}$ behaves under the extended semantics exactly as $\langle C, s\rangle$ under the original semantics.

These extended semantics of Imp are useful for different reasons: (i) They make read/write actions on the state explicitly observable as labeled transitions in the style of *Labeled Transition System* semantics for process calculi. This helps us proving a semantic correspondence in Section 3.2. (ii) Further, the extended semantics allow us to characterize the security of Imp programs using a notion of bisimulation up to high, similar to the one defined for VSPA. As a matter of fact, in Section 4, we show how security of Imp programs can be equivalently expressed in the style of *P_BNDC*, facilitating the proof that the security of Imp programs is the same as *P_BNDC* security of their translations into VSPA.

3.2 Translation

We translate Imp into VSPA following the translation described by Milner in [22], with the following important modifications: (i) We make explicit the fact that the external (possibly hostile) environment can manipulate the shared memory but cannot directly interact with a program. This is achieved by equipping registers, i.e., processes implementing Imp variables, with read/write channels accessible by the environment. All the other channels are "internalized" through restriction operators. (ii) We use a `lock` to guarantee the atomicity of expression evaluations. In fact, Imp expressions are evaluated in one atomic step. Since expression evaluation is translated into a process which sequentially accesses registers in order to read the actual variable values, to regain atomicity we need to guarantee that variables are not modified during this reading phase.

The language we want to translate contains program variables, to which values may be assigned, and the meaning of a program variable Id is a "storage location". We therefore begin by defining a storage register holding a value v as follows:

$$\begin{aligned} Reg(v) \stackrel{\text{def}}{=} & \ \texttt{put}x.Reg(x) + \overline{\texttt{get}}v.Reg(v) \\ & + \overline{\texttt{lock}}.(\texttt{eput}x.\overline{\texttt{unlock}}.Reg(x) + \overline{\texttt{unlock}}.Reg(v)) \\ & + \overline{\texttt{lock}}.(\overline{\texttt{eget}}v.\overline{\texttt{unlock}}.Reg(v) + \overline{\texttt{unlock}}.Reg(v)) \end{aligned}$$

(We shall often write $\texttt{put}(x)$ as $\texttt{put}x$ etc.) Thus, via $\overline{\texttt{get}}$ the stored value v may be read from the register, and via `put` a new value x may be written to the register. Actions $\overline{\texttt{eget}}$ and `eput` are intended to model the interactions of an external observer with the register. Before and after such actions, $\overline{\texttt{lock}}$ and $\overline{\texttt{unlock}}$ are required to be executed in order to guarantee mutual exclusion on the memory between expression evaluations and the environment. This implements the atomic expression evaluation of Imp. We also have an (abstract) time-out mechanism, that nondeterministically unlocks the registers. This is necessary to avoid blocking the program by the environment via refusing to accept $\overline{\texttt{eget}}$ or to execute `eput` after the lock has been grabbed. As a matter of fact, we want the environment to interact with the registers without interfering in any way with the program execution. The (global) lock is implemented by the process:

$$Lock \stackrel{\text{def}}{=} \texttt{lock}.\texttt{unlock}.Lock$$

For each program variable Id, we introduce a register $Reg_{Id}(y) \stackrel{\text{def}}{=} Reg(y)[g_{Id}]$, where g_{Id} is the relabeling function $\{\texttt{put}_{Id}/\texttt{put},\ \texttt{get}_{Id}/\texttt{get},\ \texttt{eput}_{Id}/\texttt{eput}, \texttt{eget}_{Id}/\texttt{eget}\}$.

This representation of registers—or program variables—as processes is fundamental to our translation; it indicates that resources like variables, as well as the programs which use them, can be thought of as processes, so that our calculus can get away with the single notion of process to represent different kinds of entity.

There is no basic notion of sequential composition in our calculus, but we can define it. To do this, we assume that processes may indicate their termination by a special channel $\overline{\texttt{done}}$. We say that a process is *well-terminating* if it cannot do any further move after performing $\overline{\texttt{done}}$; as we will see, the processes which arise from translating Imp commands are all well-terminating, since they terminate with $\overline{\texttt{done}}.\mathbf{0}$ (written $Done$) instead of just $\mathbf{0}$.

The combinator $Before$ for sequential composition is as follows:

$$P\ Before\ Q \stackrel{\text{def}}{=} (P[b/\texttt{done}]|b.Q) \setminus \{b\}$$

where b is a new name, so that no conflict arises with the done action performed by Q. It is easy to see that $Before$ preserves well-termination, i.e., if P and Q are well-terminating then so is $P\ Before\ Q$.

An expression of the language will "terminate" by yielding up its results via the special channel $\overline{\texttt{res}}$, not used by processes. If P represents such an expression, then we may wish another process Q to refer to the result by using the value variable x. To this end, we define another combinator, $Into$:

$$P\ Into(x)\ Q(x) \stackrel{\text{def}}{=} (P|\texttt{res}(x).Q(x)) \setminus \{\texttt{res}\}$$

$Q(x)$ is parametric on x and $Into$ binds this variable to the result of the expression P. Notice that we do not need to relabel res to a new channel, as we did with the special channel done. Indeed, $Q(x)$ is a process and not an expression, thus it does not use channel res to communicate with sibling processes and no conflict is ever possible. Note that $Q(x)$ might use res into a nested $Into$ combinator. In this case, however, res would be inside the scope of a restriction thus not be visible at this external $Into$ level.

The translation function $\mathcal{T}$ of Imp commands into VSPA processes is given in Fig. 4. Each expression $F(Id_1, \ldots Id_n)$ is translated into a process which collects the values of $Id_1, \ldots Id_n$ and returns $F(x_1, \ldots, x_n)$ over channel res. A state s associating variables $Id_1, \ldots, Id_m$ to values $s(Id_1), \ldots, s(Id_m)$, respectively, is translated into the parallel composition of the relative registers. The translation of commands is straightforward. Note that before and after each expression evaluation we lock and release the global lock so that the environment cannot interact with the memory while expressions are evaluated. This achieves atomic expression evaluations as in Imp. Configurations $\langle C, s\rangle$ are translated as the parallel composition of the global $Lock$ and the translations of C and s. $ACC_s = \{\overline{\texttt{put}_{Id_1}}, \texttt{get}_{Id_1}, \ldots, \overline{\texttt{put}_{Id_m}}, \texttt{get}_{Id_m}, \texttt{lock}, \texttt{unlock}\}$ is the set of all channels used by commands to access registers, plus the lock commands. Thus, the restriction over $ACC_s \cup \{\texttt{done}\}$ aims both at internalizing all the communications

$$
\begin{aligned}
\mathcal{T}[\![F(Id_1,\ldots Id_n)]\!] &= \mathtt{get}_{Id_1}x_1.\cdots.\mathtt{get}_{Id_n}x_n.\overline{\mathtt{res}}(F(x_1,\ldots,x_n)).\mathbf{0}\\
\mathcal{T}[\![s]\!] &= Reg_{Id_1}(s(Id_1))|\ldots|Reg_{Id_m}(s(Id_m))\\
\mathcal{T}[\![\mathsf{stop}]\!] &= \mathbf{0}\\
\mathcal{T}[\![\mathsf{skip}]\!] &= \overline{\mathtt{lock}}.\mathtt{tick}.\overline{\mathtt{unlock}}.Done\\
\mathcal{T}[\![Id := Exp]\!] &= \overline{\mathtt{lock}}.\mathcal{T}[\![Exp]\!]\ Into(x)\ (\overline{\mathtt{put}}_{Id}x.\mathtt{tick}.\overline{\mathtt{unlock}}.Done)\\
\mathcal{T}[\![C_1;C_2]\!] &= \mathcal{T}[\![C_1]\!]\ Before\ \mathcal{T}[\![C_2]\!]\\
\mathcal{T}[\![\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2]\!] &= \overline{\mathtt{lock}}.\mathcal{T}[\![B]\!]\ Into(x)\ (\mathbf{if}\ x\ \mathbf{then}\ \mathtt{tick}.\overline{\mathtt{unlock}}.\mathcal{T}[\![C_1]\!]\\
&\qquad\qquad \mathbf{else}\ \mathtt{tick}.\overline{\mathtt{unlock}}.\mathcal{T}[\![C_2]\!])\\
\mathcal{T}[\![\mathsf{while}\ B\ \mathsf{do}\ C]\!] &= Z \quad \text{where } Z \stackrel{\text{def}}{=} \overline{\mathtt{lock}}.\mathcal{T}[\![B]\!]\ Into(x)\ (\mathbf{if}\ x\ \mathbf{then}\ \mathtt{tick}.\\
&\qquad\qquad \overline{\mathtt{unlock}}.\mathcal{T}[\![C]\!]\ Before\ Z\ \mathbf{else}\ \mathtt{tick}.\overline{\mathtt{unlock}}.Done)\\
\mathcal{T}[\![\langle C,s\rangle]\!] &= (\mathcal{T}[\![s]\!] \mid \mathcal{T}[\![C]\!] \mid Lock) \setminus ACC_s \cup \{\mathtt{done}\}\\
\mathcal{T}[\![\langle C,s\rangle \setminus R]\!] &= \mathcal{T}[\![\langle C,s\rangle]\!] \setminus R
\end{aligned}
$$

Fig. 4. Translation of commands

between commands and registers and at removing the last done action. The environment channels $\mathtt{eput}_{Id}$ and $\mathtt{eget}_{Id}$ are not restricted and, together with tick, they are the only observable actions of $\mathcal{T}[\![\langle C,s\rangle]\!]$: $\mathtt{eput}_{Id}$ and $\mathtt{eget}_{Id}$ are high if the corresponding Imp variable Id is high; all the other observable actions, including tick, are low (the security level of unobservable actions is irrelevant for the security definition).

Examples. Consider the program $l := h$ where h is a high variable and l is a low one. These variables are represented by processes $Reg_h(s(h))$ and $Reg_l(s(l))$ for a state s.

$$
\begin{aligned}
\mathcal{T}[\![l := h]\!] =& \overline{\mathtt{lock}}.\mathcal{T}[\![h]\!]\ Into(x)\ (\overline{\mathtt{put}}_l x.\mathtt{tick}.\overline{\mathtt{unlock}}.Done)\\
=& (\overline{\mathtt{lock}}.\mathtt{get}_h x.\overline{\mathtt{res}}x.\mathbf{0} \mid \mathtt{res}(x).(\overline{\mathtt{put}}_l x.\mathtt{tick}.\overline{\mathtt{unlock}}.\overline{\mathtt{done}}.\mathbf{0})) \setminus \{\mathtt{res}\}
\end{aligned}
$$

(Notice that expression h can be seen as $ID(h)$ where ID is the identity function over Val.) The execution of the translation in a state s is as follows where $s' = [l \mapsto s(h)]s$:

$$
\begin{aligned}
&\mathcal{T}[\![\langle l := h, s\rangle]\!] = (\mathcal{T}[\![s]\!] \mid \mathcal{T}[\![l := h]\!] \mid Lock) \setminus ACC_s \cup \{\mathtt{done}\}\\
&= (\mathcal{T}[\![s]\!] \mid (\overline{\mathtt{lock}}.\mathtt{get}_h x.\overline{\mathtt{res}}x.\mathbf{0} \mid \mathtt{res}(x).(\overline{\mathtt{put}}_l x.\mathtt{tick}.\overline{\mathtt{unlock}}.\overline{\mathtt{done}}.\mathbf{0})) \setminus \{\mathtt{res}\}\\
&\quad \mid Lock) \setminus ACC_s \cup \{\mathtt{done}\} \qquad \text{(by definition of } \mathcal{T}[\![l := h]\!])\\
&\xrightarrow{\tau} (\mathcal{T}[\![s]\!] \mid (\mathtt{get}_h x.\overline{\mathtt{res}}x.\mathbf{0} \mid \mathtt{res}(x).(\overline{\mathtt{put}}_l x.\mathtt{tick}.\overline{\mathtt{unlock}}.\overline{\mathtt{done}}.\mathbf{0})) \setminus \{\mathtt{res}\}\\
&\quad \mid \mathtt{unlock}.Lock) \setminus ACC_s \cup \{\mathtt{done}\} \qquad \text{(by synchronization on lock)}\\
&\xrightarrow{\tau} (\mathcal{T}[\![s]\!] \mid (\overline{\mathtt{res}}s(h).\mathbf{0} \mid \mathtt{res}(x).(\overline{\mathtt{put}}_l x.\mathtt{tick}.\overline{\mathtt{unlock}}.\overline{\mathtt{done}}.\mathbf{0})) \setminus \{\mathtt{res}\}\\
&\quad \mid \mathtt{unlock}.Lock) \setminus ACC_s \cup \{\mathtt{done}\} \qquad \text{(fetching the value } s(h) \text{ of } h \text{ from } Reg_h)\\
&\xrightarrow{\tau} (\mathcal{T}[\![s]\!] \mid (\mathbf{0} \mid (\overline{\mathtt{put}}_l s(h).\mathtt{tick}.\overline{\mathtt{unlock}}.\overline{\mathtt{done}}.\mathbf{0})) \setminus \{\mathtt{res}\}\\
&\quad \mid \mathtt{unlock}.Lock) \setminus ACC_s \cup \{\mathtt{done}\} \qquad \text{(passing } s(h) \text{ on } res)\\
&\xrightarrow{\tau} (\mathcal{T}[\![s']\!] \mid (\mathbf{0} \mid (\mathtt{tick}.\overline{\mathtt{unlock}}.\overline{\mathtt{done}}.\mathbf{0})) \setminus \{\mathtt{res}\}\\
&\quad \mid \mathtt{unlock}.Lock) \setminus ACC_{s'} \cup \{\mathtt{done}\} \qquad \text{(updating } Reg_l \text{ with } s(h)\text{; new state is } s')\\
&\xrightarrow{\mathtt{tick}} (\mathcal{T}[\![s']\!] \mid (\mathbf{0} \mid \overline{\mathtt{unlock}}.\overline{\mathtt{done}}.\mathbf{0})) \setminus \{\mathtt{res}\}\\
&\quad \mid \mathtt{unlock}.Lock) \setminus ACC_{s'} \cup \{\mathtt{done}\} \qquad \text{(performing tick)}
\end{aligned}
$$

$$\begin{aligned}
&\xrightarrow{\tau}(\mathcal{T}[\![s']\!] \mid (\mathbf{0} \mid \overline{\texttt{done}}.\mathbf{0})) \setminus \{\texttt{res}\} \mid Lock) \setminus ACC_{s'} \cup \{\texttt{done}\} && \text{(unlocking)}\\
&\approx(\mathcal{T}[\![s']\!] \mid \mathbf{0} \mid\ Lock) \setminus ACC_{s'} \cup \{\texttt{done}\} && \text{(bisimilarity)}\\
&=\mathcal{T}[\![\langle \mathsf{stop}, s' \rangle]\!] && \text{(definition of translation)}
\end{aligned}$$

We have demonstrated that $\mathcal{T}[\![\langle l := h, s\rangle]\!] \stackrel{\hat{\texttt{tick}}}{\Longrightarrow} P$ for P such that $P \approx \mathcal{T}[\![\langle \mathsf{stop}, s'\rangle]\!]$.

As another example, the program if $h > 0$ then $l := 1$ else $l := 0$ is translated into:

$$\begin{aligned}
&\mathcal{T}[\![\mathsf{if}\ h > 0\ \mathsf{then}\ l := 1\ \mathsf{else}\ l := 0]\!]\\
&=\overline{\texttt{lock}}.\mathcal{T}[\![h > 0]\!]\ \mathit{Into}(x)\\
&\quad(\mathbf{if}\ x\ \mathbf{then}\ \texttt{tick}.\overline{\texttt{unlock}}.\mathcal{T}[\![l := 1]\!]\ \mathbf{else}\ \texttt{tick}.\overline{\texttt{unlock}}.\mathcal{T}[\![l := 0]\!])\\
&=(\overline{\texttt{lock}}.\texttt{get}_h x.\overline{\texttt{res}}(x > 0).\mathbf{0} \mid \texttt{res}(x).\\
&\qquad(\mathbf{if}\ x\ \mathbf{then}\ \texttt{tick}.\overline{\texttt{unlock}}.\mathcal{T}[\![l := 1]\!]\ \mathbf{else}\ \texttt{tick}.\overline{\texttt{unlock}}.\mathcal{T}[\![l := 0]\!])) \setminus \{\texttt{res}\}\\
&=(\overline{\texttt{lock}}.\texttt{get}_h x.\overline{\texttt{res}}(x > 0).\mathbf{0} \mid \texttt{res}(x).\\
&\qquad(\mathbf{if}\ x\ \mathbf{then}\ \texttt{tick}.\overline{\texttt{unlock}}.\\
&\qquad\quad(\overline{\texttt{lock}}.\overline{\texttt{res}}1.\mathbf{0} \mid \texttt{res}(x).(\overline{\texttt{put}}_l x.\texttt{tick}.\overline{\texttt{unlock}}.\overline{\texttt{done}}.\mathbf{0})) \setminus \{\texttt{res}\}\\
&\qquad\mathbf{else}\ \texttt{tick}.\overline{\texttt{unlock}}.\\
&\qquad\quad(\overline{\texttt{lock}}.\overline{\texttt{res}}0.\mathbf{0} \mid \texttt{res}(x).(\overline{\texttt{put}}_l x.\texttt{tick}.\overline{\texttt{unlock}}.\overline{\texttt{done}}.\mathbf{0})) \setminus \{\texttt{res}\})) \setminus \{\texttt{res}\}
\end{aligned}$$

Semantic Correspondence. The propositions below state the semantic correspondence between any R-restricted configuration $\langle C, s\rangle \setminus R$ and its translation $\mathcal{T}[\![\langle C, s\rangle \setminus R]\!]$. Let $Env = \{\overline{\texttt{eget}}_{\_}(\cdot), \texttt{eput}_{\_}(\cdot)\}$ denote the set of all the possible environment actions.

Proposition 2. *Given an R-restricted configuration $cfg = \langle C, s\rangle \setminus R$, with $R \subseteq Env$, if $cfg \xrightarrow{a} cfg'$ then there exists a process P' such that $\mathcal{T}[\![cfg]\!] \stackrel{\hat{a}}{\Longrightarrow} P'$ and $P' \approx \mathcal{T}[\![cfg']\!]$. Moreover, when $a = \texttt{tick}$ we have that $\mathcal{T}[\![cfg]\!] \stackrel{\hat{\tau}}{\Longrightarrow} \tilde{P} \xrightarrow{\texttt{tick}} P'$ and $\tilde{P} \approx \texttt{tick}.\mathcal{T}[\![cfg']\!]$.*

Intuitively, every (possibly restricted) Imp configuration move is coherently simulated by its VSPA translation, in a way that the reached process is weakly bisimilar to the translation of the reached configuration. Moreover, for `tick` moves, the translation $\mathcal{T}[\![cfg]\!]$ always reaches a state equivalent to $\texttt{tick}.\mathcal{T}[\![cfg']\!]$ before actually performing the `tick`. Intuitively, this is due to the fact that the lock is released only after `tick` is performed. Notice that if $R = \emptyset$ there is no restriction at all.

Next proposition is about the other way around: each process which is weakly bisimilar to the translation of a (possibly restricted) Imp configuration cfg always moves to processes weakly bisimilar to either $\mathcal{T}[\![cfg']\!]$ or $\texttt{tick}.\mathcal{T}[\![cfg'']\!]$, where cfg' and cfg'' are reached from cfg by performing the expected corresponding actions. As for previous proposition, $\texttt{tick}.\mathcal{T}[\![cfg'']\!]$ represents an intermediate state reached before performing the actual `tick` action.

Proposition 3. *Given an R-restricted configuration $cfg = \langle C, s\rangle \setminus R$, with $R \subseteq Env$, and a process P*

- *if $P \approx \mathcal{T}[\![cfg]\!]$ and $P \xrightarrow{\tau} P'$ then either $P' \approx P$ or $P' \approx \texttt{tick}.\mathcal{T}[\![cfg']\!]$ and $cfg \xrightarrow{\texttt{tick}} cfg'$;*
- *if $P \approx \mathcal{T}[\![cfg]\!]$ and $P \xrightarrow{a} P'$ with $a \neq \tau, \texttt{tick}$, then either $P' \approx \mathcal{T}[\![cfg']\!]$ and $cfg \xrightarrow{a} cfg'$ or $P' \approx \texttt{tick}.\mathcal{T}[\![cfg'']\!]$ and $cfg \xrightarrow{a} cfg' \xrightarrow{\texttt{tick}} cfg''$.*

4 Security Correspondence

We study the relationship between the security of Imp programs and that of VSPA processes. First, we give a notion of weak bisimulation up to high in the Imp setting, which allows us to characterize the security of Imp programs in a *P_BNDC* style. Then, we show that this new characterization of Imp program security exactly corresponds to requiring *P_BNDC* of VSPA program translations. More specifically, we prove that a program C is secure if and only if its translation $\mathcal{T}[\![\langle C, s\rangle]\!]$ is *P_BNDC* for all states s.

***P_BNDC*-Like Security Characterization for Imp.** In order to define weak bisimulation up to high, similarly to what we have done for VSPA, we define the operation $\tilde{a}$ to be a in case a is low, and a or null (which means no action) in case a is high.

Definition 7. *A symmetric binary relation R on (possibly restricted) configurations is a* bisimulation up to high *if whenever $cfg_1\ R\ cfg_2$ and $cfg_1 \xrightarrow{a} cfg_1'$, there exists cfg_2' such that $cfg_2 \xrightarrow{\tilde{a}} cfg_2'$ and $cfg_1'\ R\ cfg_2'$.*

We write $\approx_{\backslash H}$ for the union of all bisimulations up to high. This definition brings us close to the nature of the process-algebraic security specification from Section 2.2. Using bisimulation up to high and restriction we can faithfully represent the original definition of strong low-bisimulation. The following proposition states the correspondence between strong low-bisimulation (defined on the `tick` actions of the original semantics) and bisimulation up to high (defined on the extended semantics) with restriction:

Proposition 4. *$C \approx_L D$ iff $\langle C, s\rangle \approx_{\backslash H} \langle D, s\rangle \setminus H$ and $\langle C, s\rangle \setminus H \approx_{\backslash H} \langle D, s\rangle$, $\forall\ s$.*

As a direct consequence, the security of Imp can be expressed in a "*P_BNDC* style":

Corollary 1. *A program C is secure iff $\langle C, s\rangle \approx_{\backslash H} \langle C, s\rangle \setminus H$ for all s.*

Program Security is *P_BNDC*. The following theorem shows that weak bisimulation up to high is preserved in the translation from Imp to VSPA.

Theorem 1. *Let $cfg_1 = \langle C, s\rangle \setminus R$ and $cfg_2 = \langle D, t\rangle \setminus R'$, with $R, R' \subseteq H$, be two configurations (possibly) restricted over high level actions. It holds that $cfg_1 \approx_{\backslash H} cfg_2$ iff $\mathcal{T}[\![cfg_1]\!] \approx_{\backslash H} \mathcal{T}[\![cfg_2]\!]$.*

This theorem has the flavor of a full-abstraction result (cf. [1]) for the indistinguishability relation $\approx_{\backslash H}$. As a corollary of the theorem, we receive a direct link between program security and P_BNDC security:

Corollary 2. *A program C is secure iff its translation $\mathcal{T}[\![\langle C, s\rangle]\!]$ is P_BNDC $\forall\ s$.*

Examples. Recall from Section 2.1 that the program $l := h$ is rejected by the security definition for Imp. Recall from Section 3.2 that

$$\mathcal{T}[\![l := h]\!] = (\overline{\texttt{lock}}.\texttt{get}_h x.\overline{\texttt{res}}x.\mathbf{0} \mid \texttt{res}(x).(\overline{\texttt{put}}_l x.\texttt{tick}.\overline{\texttt{unlock}}.\overline{\texttt{done}}.\mathbf{0})) \setminus \{\texttt{res}\}$$

To see that this translation is rejected by *P_BNDC*, take a state s that, for example, maps all its variables to 0. We demonstrate that $\mathcal{T}[\![\langle l := h, s\rangle]\!] \setminus H \not\approx_{\backslash H} \mathcal{T}[\![\langle l := h, s\rangle]\!]$.

Varying the high variable from 0 to 1 on the right-hand side can be done by the transition $\mathcal{T}[\![\langle l := h, s\rangle]\!] \stackrel{\mathtt{eput}_h(1)}{\longrightarrow} F$ for some F. If the translation were secure then this transition would have to be simulated up to H by $\mathcal{T}[\![\langle l := h, s\rangle]\!] \setminus H$. Such a transition would have to be a $\hat{\tau}$ transition because $\mathtt{eput}_h(1)$ is a high transition, but $\mathcal{T}[\![\langle l := h, s\rangle]\!] \setminus H$ is restricted from high actions. Therefore, $\mathcal{T}[\![\langle l := h, s\rangle]\!] \setminus H$ would reduce to some process E, whose register for h remains 0.

By the definition of weak bisimulation up to H, we would have $E \setminus H \approx_{\setminus H} F$. Let subsequent actions correspond to traversing the two processes passing $\overline{\mathtt{put}}_l(0)$ and $\overline{\mathtt{put}}_l(1)$, respectively, and reaching $\overline{\mathtt{unlock}}$. Note that actions on internal channels $\mathtt{lock}, \mathtt{get}_h, \mathtt{res}, \mathtt{put}_l$ are hidden from the environment. However, as an effect of the latter action, the register for l will store different values. Even though the `tick` actions can still be simulated, this breaks bisimulation because the externally visible action $\mathtt{get}_l(0)$ by the successor of E (after $\overline{\mathtt{unlock}}$) cannot be simulated by the successor of F (after $\overline{\mathtt{unlock}}$).

Further, recall from Section 2.1 that the program if $h > 0$ then $l := 1$ else $l := 0$ is rejected by Imp's security definition. In Section 3.2 we saw that

$$
\begin{array}{l}
\mathcal{T}[\![\textsf{if } h > 0 \textsf{ then } l := 1 \textsf{ else } l := 0]\!] = \\
\quad (\overline{\mathtt{lock}}.\mathtt{get}_h x.\overline{\mathtt{res}}(x > 0).\mathbf{0} \mid \mathtt{res}(x). \\
\quad\quad (\textbf{if } x \textbf{ then } \mathtt{tick}.\overline{\mathtt{unlock}}. \\
\quad\quad\quad (\overline{\mathtt{lock}}.\overline{\mathtt{res}}1.\mathbf{0} \mid \mathtt{res}(x).(\overline{\mathtt{put}}_l x.\mathtt{tick}.\overline{\mathtt{unlock}}.\overline{\mathtt{done}}.\mathbf{0})) \setminus \{\mathtt{res}\} \\
\quad\quad \textbf{else } \mathtt{tick}.\overline{\mathtt{unlock}}. \\
\quad\quad\quad (\overline{\mathtt{lock}}.\overline{\mathtt{res}}0.\mathbf{0} \mid \mathtt{res}(x).(\overline{\mathtt{put}}_l x.\mathtt{tick}.\overline{\mathtt{unlock}}.\overline{\mathtt{done}}.\mathbf{0})) \setminus \{\mathtt{res}\})) \setminus \{\mathtt{res}\}
\end{array}
$$

The information flow from $h > 0$ to l is evident in the translation. The result of inspecting the expression $h > 0$ is sent on the channel `res`. When this result is received and checked, either it triggers the process that puts 1 in the register for l or a similar process that puts 0 to that register.

As above, the VSPA translation fails to satisfy *P_BNDC*. Varying the high state by a high environment action $\mathtt{eput}_h(\cdot)$ in the beginning leads to different values in the register for l. This difference can be observed by low environment actions $\overline{\mathtt{eget}}_l(\cdot)$.

5 Related Work

A large body of work on information-flow security has been developed in the area of programming languages (see a recent survey [27]) and process calculus (e.g., [7, 25, 24, 13, 18]). While both language-based and process calculus-based security are relatively established fields, only little has been done for understanding the connection between the two.

A line of work initiated by Honda et al. [14] and pursued by Honda and Yoshida [15, 16] develops security type systems for the π-calculus. The use of linear and affine types gives the power for these systems to soundly embed type systems for imperative multi-threaded languages [29] into the typed π-calculus. This direction is appealing as it leads to automatic security enforcement mechanisms by security type checking. Nevertheless, automatic enforcement comes at the price of lower precision. Our approach opens

up possibilities for combining high-precision security verification (such as equivalence checking in process calculi [23]) with type-based verification. Steps in this direction have been made in, e.g., [17, 2, 31], however, not treating timing-sensitive security.

Giambiagi and Dam's work on *admissible flows* [3, 11] illustrates a useful synergy of an imperative language and a CCS-like process calculus. The assurance provided by admissible flows is that a security protocol implementation (written in the imperative language) leaks no more information than prescribed in the specification (written in the process calculus).

Mantel and Sabelfeld [21] have suggested an embedding of a multithreaded and distributed language into MAKS [20], an abstract framework for modeling the security of event-based systems. The translation of a program is secure (as an event system) if and only if the program itself is secure (in the sense that the program satisfies self-low-similarity). While this work offers a useful connection between language-based and event-based security, it is inherently restricted to expressing event systems as trace models. In the present work, the security of both the source and target languages is defined in terms of bisimulation. This enables us to capture additional information leaks, e.g., through deadlock behavior [7], which trace-based models generally fail to detect.

Our inspiration for handling timing-sensitive security stems from the work by Focardi et al. [8], where explicit `tick` events are used to keep track of timing in a scenario of a discrete-time process calculus.

6 Conclusion and Future Work

We have established a formal connection between a language-based and a process calculus security definition of information security. Concretely, we have shown that a timing-sensitive security definition corresponds to P_BNDC, persistent bisimulation-based nondeducibility on compositions. Thereby, we have identified a point in the space of process calculus-based definitions [7] that exactly corresponds to compositional timing-sensitive language-based security.

Drawing on Milner's work [22], we have developed a generally useful encoding of an imperative language into a CCS-like calculus. We expect that this encoding will be helpful for both future work on information security topics as well as other topics that necessitate representation of programming languages in process calculus.

This paper sets solid ground for future work in the following directions:

Security policies: We have used as a starting point a timing-sensitive language-based security specification. This choice has allowed us to establish a tight, timing-sensitive, correspondence between computation steps in the imperative language and the actions of processes. However, it is important to consider a full spectrum of attackers, including the attacker that may not observe (non)termination. Future work includes weakening security policies and investigating the relation between the two kinds of security for a termination-insensitive attacker.

Concurrency and distribution: Concurrency and distribution are out of scope for this paper for lack of space. However, the technical machinery is already in place to add multithreading and distribution to the imperative language (for example, the program security characterization is known to be compositional for Imp with dynamic thread

creation [28]). We conjecture that in presence of concurrency, P_BNDC will remain to correspond to the language-based security definition. We expect parallel compositions of Imp threads to be encoded by parallel compositions of VSPA processes. In this case, the security correspondence result would be a consequence of the compositionality of the two properties. We anticipate the security correspondence to hold without major changes in the encoding. The effect of distribution features in both source and target languages is certainly a worthwhile topic for future work. An extension of the source language with channel-based communication [26] is a natural point for investigating the connection to process calculi security. As a matter of fact, P_BNDC has been specifically developed for communicating processes, thus it should be applicable even when channels are used both for communication and for manipulating memories.

Modular security: According to the vision we stated in the introduction, for the security analysis of heterogeneous systems we need heterogeneous, scalable techniques. The key to scalability is modular analysis that allows analyzing parts of a systems in isolation and plug together secure components together. That the resulting system is secure is guaranteed by compositionality results. While compositionality properties for Imp and VSPA have been studied separately, we intend to explore the interplay between the two. In particular, we expect to obtain stronger compositionality results for the image of secure imperative programs in VSPA than for regular VSPA processes.

References

1. M. Abadi. Protection in programming-language translations. In *Proc. International Colloquium on Automata, Languages, and Programming*, volume 1443 of *LNCS*, pages 868–883. Springer-Verlag, July 1998.
2. D. Clark, C. Hankin, and S. Hunt. Information flow for Algol-like languages. *Journal of Computer Languages*, 28(1):3–28, April 2002.
3. M. Dam and P. Giambiagi. Confidentiality for mobile code: The case of a simple payment protocol. In *Proc. IEEE Computer Security Foundations Workshop*, pages 233–244, July 2000.
4. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.
5. R. Focardi and R. Gorrieri. A Classification of Security Properties for Process Algebras. *Journal of Computer Security*, 3(1):5–33, 1994/1995.
6. R. Focardi and R. Gorrieri. The Compositional Security Checker: A Tool for the Verification of Information Flow Security Properties. *IEEE Transactions on Software Engineering*, 23(9):550–571, 1997.
7. R. Focardi and R. Gorrieri. Classification of Security Properties (Part I: Information Flow). In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*, pages 331–396. Springer-Verlag, 2001.
8. R. Focardi, R. Gorrieri, and F. Martinelli. Information flow analysis in a discrete-time process algebra. In *Proc. IEEE Computer Security Foundations Workshop*, pages 170–184, July 2000.
9. R. Focardi and S. Rossi. Information Flow Security in Dynamic Contexts. In *Proc. of the IEEE Computer Security Foundations Workshop*, pages 307–319. IEEE Computer Society Press, 2002.

10. R. Focardi, S. Rossi, and A. Sabelfeld. Bridging Language-Based and Process Calculi Security. Technical Report CS-2004-14, Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy, 2004. http://www.dsi.unive.it/ricerca/TR/index.htm.
11. P. Giambiagi and M.Dam. On the secure implementation of security protocols. In *Proc. European Symp. on Programming*, volume 2618 of *LNCS*, pages 144–158. Springer-Verlag, April 2003.
12. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.
13. M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM TOPLAS*, 24(5):566–591, 2002.
14. K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proc. European Symp. on Programming*, volume 1782 of *LNCS*, pages 180–199. Springer-Verlag, 2000.
15. K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 81–92, January 2002.
16. K. Honda and N. Yoshida. Noninterference through flow analysis. *Journal of Functional Programming*, 2005. To appear.
17. R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.
18. N. Kobayashi. Type-based information flow analysis for the pi-calculus. Technical Report TR03-0007, Tokyo Institute of Technology, October 2003.
19. B. W. Lampson. A note on the confinement problem. *Comm. of the ACM*, 16(10):613–615, October 1973.
20. H. Mantel. Possibilistic definitions of security – An assembly kit –. In *Proc. IEEE Computer Security Foundations Workshop*, pages 185–199, July 2000.
21. H. Mantel and A. Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *J. Computer Security*, 11(4):615–676, September 2003.
22. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
23. C. Piazza, E. Pivato, and S. Rossi. CoPS - Checker of Persistent Security. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 144–152. Springer-Verlag, March 2004.
24. F. Pottier. A simple view of type-secure information flow in the pi-calculus. In *Proc. IEEE Computer Security Foundations Workshop*, pages 320–330, June 2002.
25. P. Ryan. Mathematical models of computer security—tutorial lectures. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*, pages 1–62. Springer-Verlag, 2001.
26. A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proc. Symp. on Static Analysis*, volume 2477 of *LNCS*, pages 376–394. Springer-Verlag, September 2002.
27. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
28. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.
29. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, January 1998.
30. G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, 1993.
31. N. Yoshida, K. Honda, and M. Berger. Linearity and bisimulation. In *Proc. Foundations of Software Science and Computation Structure*, volume 2303 of *LNCS*, pages 417–433. Springer-Verlag, April 2002.