Dynamics and Secure Information Flow for a Higher-Order Pi-Calculus *

Martin Pettai[1,2] and Peeter Laud[2]

[1] University of Tartu
[2] Cybernetica AS

Abstract We show how a type system for secure information flow for a π-calculus with higher-order λ-abstractions can be extended with dynamics without weakening the non-interference guarantees. The type system for the π-calculus ensures that the traffic on high channels does not influence the traffic on low channels. λ-abstractions make it possible to send processes over channels. Dynamics make it possible to send processes and other data of different types over the same channel, making communication between processes easier. If dynamics are used, the types of some expressions or channels may depend on type variables that are instantiated at run time. To make it still possible to statically check secure information flow, we ensure that instantiating a type variable in an expression also instantiates it in the type of the expression.

1 Introduction

The question of information security arises when the inputs and outputs of a program are partitioned into different security classes. In this case we want the high-security inputs not inappropriately influence the low-security outputs and other behaviour observable at low clearance. The strongest such property is *non-interference* [10] stating that there is no influence at all; or that variations in the high-security inputs do not change the observations at the low level.

Over the years, static analyses, typically type systems for verifying secure information flow have been proposed for programs written in many kinds of programming languages and paradigms — imperative or functional, sequential or parallel, etc. Each new construct in the language can have a profound effect on the information flows the programs may have. For a language to be usable in practice, it usually needs to have many different constructs, which makes information flow analysis much more complicated than in simple languages. In spite of this, there exist some practical languages with information flow type

* This research was supported by the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS, and the Software Technologies and Applications Competence Centre, STACC. This research was also supported by Personal Research Grant PUT2 by Estonian Research Council, and by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 231620.

systems, such as Jif, which is based on JFlow [13]. Thus typing is a practical way of checking secure information flow.

In strongly typed functional programming languages, static typing is widely used to guarantee type safety. In some cases, however, type information only becomes available at run time. For example, data may be obtained from the network or from user input. In the case of mobile code, which is becoming ubiquitous, also code is obtained from the network. In these cases, the values (data or code) may be wrapped in black boxes called *dynamics* or dynamic values [2] that in addition to a value also contain the type of this value. The types in these black boxes can be compared at run time with each other or with statically known types. If several types are allowed then run-time branching on the types can be used to exhibit different behavior for different types. In each branch, enough type information is known statically, so static typing can again be used to type check the individual branches.

In this paper, we will see that dynamics can be used not only in an ordinary type system but also in a type system for secure information flow. Here the types wrapped in dynamics can contain security levels.

Dynamics can be useful in distributed systems where processes send messages of different types to each other. If messages are wrapped in dynamics by the sender then a single channel can be used for communication between two processes, instead of a separate channel for each possible message type. The receiver of a message can check that the type wrapped in the dynamic is one of the types that it expects and can act according to the type. To model such communication, we use the π-calculus as the base of our language (a good introduction to the π-calculus can be found in [14]). To be able to also send code between processes, not only names, we include in the language λ-expressions that can also return a process. This makes the π-calculus higher-order. Finally, we can add dynamics to the language.

Let us consider an example. Suppose we have a server that contains some public and some secret data. The server accepts queries that may need to use the public or secret data but that may also write data to public or secret channels. We want to ensure that if a query accesses a public channel then it cannot use the secret data, which might be leaked to the public channel. Here is the pseudocode:

- server:
  - variables *pubdata*, *secdata*
  - listen to channel *serv* and for each message *query* that is received:
    - if *query* contains a procedure that uses public channels then execute query with *pubdata* as an argument
    - if *query* contains a procedure that does not use public channels then execute *query* with *pubdata* and *secdata* as arguments
- client1:
  - send to *serv* a query that writes *pubdata* to a global public channel
- client2:
  - send to *serv* a query that writes *secdata* to a global secret channel

In Sec. 2.2, we will implement this example in our language. Our type system will guarantee that no information about *secdata* is leaked to a public channel.

Our goal was to have non-interference for our language. To make it easier to achieve this, we take as a base [15], which has the necessary framework needed to prove non-interference for the π-calculus. The advantage of the framework is the use of $\langle\pi\rangle$-calculus, which allows the bisimulation relation needed for non-interference to be derived naturally, instead of having to define it ad hoc. We adapt the definitions, lemmas, and theorems to our language and extend the proofs with cases corresponding to the added constructs in our language.

We will begin in Sec. 2 by introducing the syntax and run-time semantics of our language and continue by discussing an example. We will then see how a (weak) bisimulation relation arises naturally from a certain projection function. This is asserted by the lemmas that we will prove. In Sec. 3, we will introduce our type system for secure information flow and prove that the type of an expression is retained during reduction. In Sec. 4, we will state the non-interference results. The proofs for our language are essentially the same as for the language in [15] and we will not repeat them. In Sec. 5, we will see more examples. We will review the related work in Sec. 6 and discuss our results in Sec. 7.

2 Syntax and Operational Semantics

2.1 Description

Our language is based on the $\langle\pi\rangle$-calculus, which is defined in [15], augmented with (recursive) λ-expressions (with a built-in fixpoint operator as in [16]). We have added dynamics to the language, which allow introducing run-time type variables using pattern matching. We also allow sending arbitrary values along channels, not only channel names.

The $\langle\pi\rangle$-calculus adds to the π-calculus a bracket construct $\langle\mathbf{e}\rangle_i$ (where $i \in \{1,2\}$) that allows packing two high-security expressions into one to facilitate reasoning about bisimilar expressions. This construct is not meant to be used in actual programs written in the language. In the following, we use boldface meta-variables (e.g. $\mathbf{e}$ or $\mathbf{N}$) to denote expressions that do not contain this construct. Such expressions are called *standard* expressions.

The syntax of our language is given in Fig. 1 and the operational semantics in Fig. 2. We have the following three definitions (from [15]):

Definition 1. *Let* $\{i,j\} = \{1,2\}$. *The* i^{th} projection *function, written* π_i, *satisfies the laws* $\pi_i(\langle\mathbf{e}\rangle_i) = \mathbf{e}$ *and* $\pi_i(\langle\mathbf{e}\rangle_j) = \mathbf{0}$ *and is a homomorphism on standard expression forms.*

Definition 2. Structural congruence $\equiv$ *is the smallest reflexive, compatible relation over expressions which satisfies the following laws:*

1. $N+\mathbf{0} \equiv N$, $N \equiv N+\mathbf{0}$, $N_1+N_2 \equiv N_2+N_1$, $(N_1+N_2)+N_3 \equiv N_1+(N_2+N_3)$;
2. $N \mid \mathbf{0} \equiv N$, $N \equiv N \mid \mathbf{0}$, $N_1 \mid N_2 \equiv N_2 \mid N_1$, $(N_1 \mid N_2) \mid N_3 \equiv N_1 \mid (N_2 \mid N_3)$;
3. $\nu x_1 : t_1.\ \nu x_2 : t_2.\ e \equiv \nu x_2 : t_2.\ \nu x_1 : t_1.\ e$.

Definition 3. *The raw one-step reduction relation* $\longrightarrow$ *is given by Fig. 2. We write* $N_1 \# N_2$ *(read:* N_1 *and* N_2 *may communicate) for* $\exists e.(N_1 \mid N_2 \longrightarrow e \vee N_2 \mid N_1 \longrightarrow e)$*. Weak reduction, written* $\Longrightarrow$*, is defined as* $(\equiv \cup \longrightarrow)^*$*.*

We identify expressions up to α-conversion, to facilitate variable substitutions. In the following, we use an overline to denote a list of something, e.g.

$$
\begin{aligned}
t &::= \langle \bar{t} \rangle_l^p \mid \mathsf{Dynamic} \mid t \to \tau \mid \gamma \\
\tau &::= t \mid \mathsf{Proc}_{pc} \\
p &::= - \mid + \mid \pm \mid \beta \\
l, pc &::= L \mid H \mid \alpha \\
v &::= x \mid \mathsf{wrap}\ v : t \mid \mathsf{fix}\ f.\lambda x : t.\ e \\
e &::= v \mid P \mid v\ v \mid \mathsf{bind}\ x = e\ \mathsf{in}\ e \mid v\ \mathsf{unwrap}\ x :\prec t \succ e\ \mathsf{else}\ e \mid \\
&\quad \mid v\ \mathsf{unwrap}\ x : t \succ e\ \mathsf{else}\ e \\
N &::= x(\tilde{y}).\ e \mid \bar{x}\,\langle \tilde{v} \rangle\,.\ e \mid \mathbf{0} \mid N + N \\
P &::= N \mid (e \mid e) \mid !e \mid \nu x : t.\ e \mid \langle \mathbf{e} \rangle_i
\end{aligned}
$$

Figure 1. Syntax and types

$\overline{\alpha_j \Leftarrow \ell_j}$ is a list that contains the substitution $\alpha_j \Leftarrow \ell_j$ for each j in some set of indices. For a single variable without an index, we use a tilde instead of an overline to avoid confusion with sending on a channel, e.g. $\tilde{y}$ is a list of variables.

An expression (denoted by e) can reduce either to a value (denoted by v) without making any side effects, or to a procedure (denoted by P), whose further reduction may cause side effects but cannot return a value.

Value-level variables are denoted by x (sometimes also y) or f. We use f for variables that are used for recursive function calls (e.g. in $\mathsf{fix}\ f.\lambda x : t.\ f\ x$ the subexpression $f\ x$ is a recursive call with the argument x) but such variables are not syntactically distinguished from ordinary variables. Function applications are handled by the rule (app). To allow recursion, the rule replaces the variable f by a copy of the function. This recursion may be non-terminating.

A similar construct, called *replication*, is available only for procedures, not arbitrary expressions. The procedure $!P$ allows creating an arbitrary number of threads (rule (repl)), each executing P.

We distinguish value types (which we call just *types* and denote by t) and procedure types (denoted by Proc_{pc}). Extended types (denoted by τ) can be either types or procedure types.

The form of function types $t \to \tau$ shows that functions can return both values and procedures but can receive as an argument only values. If we want to give a procedure P as an argument to a function then we can use the function $\mathsf{fix}\ f.\lambda x : \mathit{Dynamic}.\ P$ (with a dummy argument x; the variable f is also not used) instead.

$$E[e] ::= \mathsf{bind}\ x = e\ \mathsf{in}\ e' \mid (e \mid e') \mid (e' \mid e) \mid \nu x : t.\ e \mid \langle e \rangle_i$$

$$(\mathsf{fix}\ f.\lambda x : t.\ e)\ v \longrightarrow e[f \Leftarrow (\mathsf{fix}\ f.\lambda x : t.\ e), x \Leftarrow v] \ (\text{app})$$

$$\mathsf{bind}\ x = v\ \mathsf{in}\ e \longrightarrow e[x \Leftarrow v] \ (\text{bind})$$

$$\frac{t_1 \leq t_2}{(\mathsf{wrap}\ v : t_1)\ \mathsf{unwrap}\ x :< t_2 \succ e_1\ \mathsf{else}\ e_2 \longrightarrow e_1[x \Leftarrow v]} \ (\text{unwrap-subt})$$

$$\frac{\neg(t_1 \leq t_2)}{(\mathsf{wrap}\ v : t_1)\ \mathsf{unwrap}\ x :< t_2 \succ e_1\ \mathsf{else}\ e_2 \longrightarrow e_2} \ (\text{unwrap-subt-else})$$

$$\frac{t_1 = t_2[\overline{\alpha_j \Leftarrow \ell_j}, \overline{\beta_j \Leftarrow p_j}, \overline{\gamma_j \Leftarrow t_j}]}{(\mathsf{wrap}\ v : t_1)\ \mathsf{unwrap}\ x : t_2 \succ e_1\ \mathsf{else}\ e_2 \longrightarrow} \ (\text{unwrap-pat})$$
$$\longrightarrow e_1[x \Leftarrow v, \overline{\alpha_j \Leftarrow \ell_j}, \overline{\beta_j \Leftarrow p_j}, \overline{\gamma_j \Leftarrow t_j}]$$

$$\frac{\neg\exists(\overline{\alpha_j}, \overline{\beta_j}, \overline{\gamma_j}).\ t_1 = t_2[\overline{\alpha_j \Leftarrow \ell_j}, \overline{\beta_j \Leftarrow p_j}, \overline{\gamma_j \Leftarrow t_j}]}{(\mathsf{wrap}\ v : t_1)\ \mathsf{unwrap}\ x : t_2 \succ e_1\ \mathsf{else}\ e_2 \longrightarrow e_2} \ (\text{unwrap-pat-else})$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \ (\text{context}) \qquad \frac{x \notin \mathrm{fn}(e_2)}{(\nu x : t.\ e_1) \mid e_2 \longrightarrow \nu x : t.\ (e_1 \mid e_2)} \ (\text{extr})$$

$$(N_1 + x(\tilde{y}).\ e_1) \mid (N_2 + \bar{x}\,\langle \tilde{v} \rangle\,.\ e_2) \longrightarrow e_1[\tilde{y} \Leftarrow \tilde{v}] \mid e_2 \ (\text{comm})$$

$$\mathbf{0} \longrightarrow \nu x : t.\ \mathbf{0} \ (\text{new}) \qquad !e \longrightarrow e \mid !e \ (\text{repl})$$

$$\frac{\pi_i N_0 \# \mathbf{N}_1 \qquad \{i, j\} = \{1, 2\}}{N_0 \mid \langle \mathbf{N}_1 \rangle_i \longrightarrow \langle \pi_i N_0 \mid \mathbf{N}_1 \rangle_i \mid \langle \pi_j N_0 \rangle_j} \ (\text{split}) \qquad \frac{\mathbf{N}_1 \# \mathbf{N}_2}{\langle \mathbf{N}_1 \rangle_i \mid \langle \mathbf{N}_2 \rangle_i \longrightarrow \langle \mathbf{N}_1 \mid \mathbf{N}_2 \rangle_i} \ (\text{glue})$$

$$\langle \mathbf{e}_1 \mid \mathbf{e}_2 \rangle_i \longrightarrow \langle \mathbf{e}_1 \rangle_i \mid \langle \mathbf{e}_2 \rangle_i \ (\text{break}) \qquad \langle \nu x : t.\ \mathbf{e} \rangle_i \longrightarrow \nu x : t.\ \langle \mathbf{e} \rangle_i \ (\text{push})$$

Figure 2. Operational semantics

The channel type $\langle \tilde{t} \rangle_l^p$ shows that a channel can be used to transmit a list of values (with types given by the list $\tilde{t}$). The channel can be used only in the context with level l. We have two security levels: low (public) and high (secret), denoted by L and H, respectively. The polarity p can be $-$ (the channel can be read), $+$ (it can be written to), or $\pm$ (it can be both read and written to).

We also have the type Dynamic, whose values can be constructed with the construct $\mathsf{wrap}\ v : t$, which wrap the value v and its type t in a single run-time value (called a *dynamic*). The type t can be inferred automatically during type checking, so the programmer may actually write just $\mathsf{wrap}\ v$.

The values of type Dynamic can be analyzed using the unwrap constructs, which allow branching according to the type contained in the dynamic. The first variant of unwrap (with $x :< t$) chooses the first branch (rule (unwrap-subt)) if and only if the type in the dynamic is a subtype of t (i.e. x can be given the type t), and the second branch (rule (unwrap-subt-else)) otherwise. The second variant (with $x : t$) uses t as a type pattern that can contain variables. The first branch is chosen (rule (unwrap-pat)) if and only if the type in the dynamic matches the pattern, and the type variables are replaced (at run time) with concrete types (or parts of types) in this case. If the pattern match fails, the second branch (rule (unwrap-pat-else)) is chosen.

We introduce three kinds of type pattern variables: α denotes variables that correspond to a security level (denoted by l or pc), β denotes variables corresponding to a polarity (denoted by p), and γ denotes variables corresponding to a (value) type.

We also have a construct $\mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2$ that fixes the order of evaluation of e_1 and e_2: first, e_1 is evaluated (using the rule (context)), then its value can be used to replace the variable x in e_2 (rule (bind)). This ensures a call-by-value semantics.

We also have the standard π-calculus constructs of sending $(\bar{x}\,\langle\tilde{v}\rangle\,.\ e)$ and receiving $(x(\tilde{y}).\ e)$ values over channel x (rule (comm)), null process ($\mathbf{0}$), sum of processes $(N + N)$, parallel composition $(e \mid e)$, and creating $(\nu x : t.\ e)$ a new channel of type t (rule (extr)). The rules (new), (split), (glue), (break), and (push) are used only for the $\langle\pi\rangle$-calculus expressions with brackets, and are similar to those in [15]. Some forms of processes (called *normal* processes) are denoted by N instead of P to facilitate the definition of operational semantics.

2.2 Example from the Introduction

We will now see how to implement the example from the introduction in our language. The code is given in Fig. 3. There we first create a global public channel, a global secret channel, and another global public channel that will be listened by the server. Then we create three threads—the server and the two clients.

The server contains some public and secret data ($SomePublicValue$ and $SomeSecretValue$, which should be expressions of type Dynamic), which are written to channels (because we do not have mutable variables). The server listens to the channel $serv$ (using replication, so that it can handle more than one query), and when it receives a query (which is contained in a Dynamic), it branches accrording to the type of the query. If the query contains a high procedure expecting two arguments then it executes the procedure with the public and secret data as the arguments. If it contains a low procedure expecting one argument then it executes the procedure with the public data as the argument.

The first client creates a query that writes the public data in the server to a global public channel. The second client creates a query that writes the secret data in the server to a global secret channel. Both clients wrap their queries into a dynamic and send them to the channel $serv$.

The server gives the secret data to a secret procedure and the public data to a public procedure as channel names, not as actual values. This allows the procedures in the query to also change the data in the server, not only read it, although the current example does not use this possibility. The public data is given to a secret procedure as an ordinary value because a secret procedure must not affect public data. To use the value in a channel, it is first read from the channel and then immediately written back to it so that it can be read again later. The write is done in a separate thread, so that the synchronous write would not block the current thread. To change the value in the channel, a different value would be written back instead of the one that was just read.

$$
\begin{array}{l}
\nu pubchan : \langle \mathsf{Dynamic} \rangle^{\pm}_{L} \,.\, \nu secchan : \langle \mathsf{Dynamic} \rangle^{\pm}_{H} \,.\, \nu serv : \langle \mathsf{Dynamic} \rangle^{\pm}_{L} \,. \\
((\\
\quad \nu pubdata : \langle \mathsf{Dynamic} \rangle^{\pm}_{L} \,.\, \nu secdata : \langle \mathsf{Dynamic} \rangle^{\pm}_{H} \,.\, (\\
\quad\quad \overline{pubdata}\,\langle SomePublicValue \rangle \,.\, \mathbf{0} \mid \\
\quad\quad \overline{secdata}\,\langle SomeSecretValue \rangle \,.\, \mathbf{0} \mid \\
\quad\quad !serv(query). \\
\quad\quad\quad query \ \mathsf{unwrap}\ q : (\mathsf{Dynamic} \to \langle \mathsf{Dynamic} \rangle^{\pm}_{H} \to \mathsf{Proc}_{H}) \succ \\
\quad\quad\quad\quad\quad pubdata(pubd). \ (\overline{pubdata}\,\langle pubd \rangle \,.\, \mathbf{0} \mid q \ pubd \ secdata) \\
\quad\quad\quad\quad \mathsf{else}\ query \ \mathsf{unwrap}\ q : (\langle \mathsf{Dynamic} \rangle^{\pm}_{L} \to \mathsf{Proc}_{L}) \succ q \ pubdata \\
\quad\quad\quad\quad \mathsf{else}\ \mathbf{0}) \\
) \mid (\\
\quad \mathsf{bind}\ queryL = \mathsf{fix}\ f.\lambda pub : \langle \mathsf{Dynamic} \rangle^{\pm}_{L} \,. \\
\quad\quad\quad pub(pubd). \ (\overline{pub}\,\langle pubd \rangle \,.\, \mathbf{0} \mid \overline{pubchan}\,\langle pubd \rangle \,.\, \mathbf{0}) \ \mathsf{in} \\
\quad\quad \overline{serv}\,\langle \mathsf{wrap}\ queryL : \langle \mathsf{Dynamic} \rangle^{\pm}_{L} \to \mathsf{Proc}_{L} \rangle \,.\, \mathbf{0} \\
) \mid (\\
\quad \mathsf{bind}\ queryH = \mathsf{fix}\ f.\lambda pubd : \mathsf{Dynamic}.\ \mathsf{fix}\ f.\lambda sec : \langle \mathsf{Dynamic} \rangle^{\pm}_{H} \,. \\
\quad\quad\quad sec(secd). \ (\overline{sec}\,\langle secd \rangle \,.\, \mathbf{0} \mid \overline{secchan}\,\langle secd \rangle \,.\, \mathbf{0}) \ \mathsf{in} \\
\quad\quad \overline{serv}\,\langle \mathsf{wrap}\ queryH : \mathsf{Dynamic} \to \langle \mathsf{Dynamic} \rangle^{\pm}_{H} \to \mathsf{Proc}_{H} \rangle \,.\, \mathbf{0}))
\end{array}
$$

Figure 3. The example from the introduction in our language

2.3 Lemmas

We will now see how an expression containing brackets is related to the two standard expressions that are packed into it. We first have an auxiliary lemma.

Lemma 1. *Let* $i \in \{1, 2\}$. *Then* $(\pi_i e)[\tilde{y} \Leftarrow \pi_i \tilde{v}] = \pi_i(e[\tilde{y} \Leftarrow \tilde{v}])$ *and* $(\pi_i e)[\tilde{y} \Leftarrow \pi_i \tilde{v}, \overline{\alpha_j \Leftarrow \ell_j}, \overline{\beta_j \Leftarrow p_j}, \overline{\gamma_j \Leftarrow t_j}] = \pi_i(e[\tilde{y} \Leftarrow \tilde{v}, \alpha_j \Leftarrow \ell_j, \beta_j \Leftarrow p_j, \gamma_j \Leftarrow t_j])$.

Proof. $\pi_i e$ has the same structure has e, except some subexpressions may have been replaced by $\mathbf{0}$. In $e[\tilde{y} \Leftarrow \tilde{v}, \overline{\alpha_j \Leftarrow \ell_j}, \overline{\beta_j \Leftarrow p_j}, \overline{\gamma_j \Leftarrow t_j}]$, substitutions have been made in all subexpressions. If we apply π_i then the substitutions are still visible in those subexpressions that were not replaced by $\mathbf{0}$ and π_i is also applied to those subexpressions that were introduced by the substitution (i.e. $\tilde{v}$, which changes to $\pi_i \tilde{v}$; the type-level expressions $\overline{\ell_j}$, $\overline{p_j}$, $\overline{t_j}$ are not affected by π_i). Thus, we get $(\pi_i e)[\tilde{y} \Leftarrow \pi_i \tilde{v}, \overline{\alpha_j \Leftarrow \ell_j}, \overline{\beta_j \Leftarrow p_j}, \overline{\gamma_j \Leftarrow t_j}]$. □

Now we can prove two lemmas that together show that e and $\pi_i e$ (considered modulo addition and removal of null processes) are weakly bisimilar.

Lemma 2. *Let* $i \in \{1, 2\}$. *If* $e \longrightarrow e'$ *then* $\pi_i e \Longrightarrow \pi_i e'$.

Proof. By induction on the derivation of $e \longrightarrow e'$. The cases corresponding to π-calculus-related constructs ((comm), (extr), (new), (repl), (split), (glue), (break), and (push)) are handled similarly to the proof of the corresponding lemma in [15].

Case (app). π_i is a homomorphism on all expression forms involved. The result follows by (app) and Lemma 1.

Case (bind). π_i is a homomorphism on all expression forms involved. $(\pi_i e)[x \Leftarrow \pi_i v]$ is $\pi_i(e[x \Leftarrow v])$ by Lemma 1. The result follows by (bind).

Case (unwrap-subt). π_i is a homomorphism on all expression forms involved. $(\pi_i e_1)[x \Leftarrow \pi_i v]$ is $\pi_i(e_1[x \Leftarrow v])$ by Lemma 1. The result follows by (unwrap-subt) (the premise $t_1 \leq t_2$ remains unchanged because types cannot contain brackets).

Case (unwrap-subt-else). π_i is a homomorphism on all expression forms involved.

Case (unwrap-pat). π_i is a homomorphism on all expression forms involved. $(\pi_i e_1)[x \Leftarrow \pi_i v, \overline{\alpha_j \Leftarrow \ell_j}, \overline{\beta_j \Leftarrow p_j}, \overline{\gamma_j \Leftarrow t_j}]$ is $\pi_i(e_1[x \Leftarrow v, \overline{\alpha_j \Leftarrow \ell_j}, \overline{\beta_j \Leftarrow p_j}, \overline{\gamma_j \Leftarrow t_j}])$ by Lemma 1. The result follows by (unwrap-pat) (the premise remains unchanged because types, polarities, and levels cannot contain brackets).

Case (unwrap-pat-else). π_i is a homomorphism on all expression forms involved.

Case (context). The subcases $E = \mathsf{bind}\ x = [] \ \mathsf{in}\ e_2$, $E = ([] \mid e_2)$, $E = (e_2 \mid [])$, and $E = \nu x : t.\ []$ can be handled by using the induction hypothesis, applying (context), and using the fact that π_i is a homomorphism on all expression forms involved. Let us now consider the subcase $E = \langle [] \rangle_j$. Then e and e' are of the form $\langle e_1 \rangle_j$ and $\langle e_2 \rangle_j$, respectively, and $e_1 \longrightarrow e_2$ holds. If $i \neq j$ then $\pi_i e = \pi_i e' = \mathbf{0}$ and the result is immediate. If $i = j$ then $\pi_i e = e_1$ and $\pi_i e' = e_2$, so $\pi_i e$ reduces to $\pi_i e'$ as desired. □

Definition 4. *Let $\leq_0$ be the smallest reflexive, compatible relation over expressions which satisfies the law $e \leq_0 e \mid \mathbf{0}$.*

Lemma 3. *Let $i \in \{1, 2\}$. If $\mathbf{e} \longrightarrow \mathbf{e}'$ and $\mathbf{e} = \pi_i e$ then there exists some expression e' such that $e \Longrightarrow e'$ and $\mathbf{e}' \leq_0 \pi_i e'$.*

Proof. By induction on the derivation of $\mathbf{e} \longrightarrow \mathbf{e}'$. The cases (comm), (extr), (repl), and (new) are handled similarly to the proof of the corresponding lemma in [15].

Before the following cases, we now consider the case where $e = \langle \mathbf{e} \rangle_i$. As $\langle \mathbf{e} \rangle_i \longrightarrow \langle \mathbf{e}' \rangle_i$ by (context), and $\mathbf{e}' = \pi_i \langle \mathbf{e}' \rangle_i$, we can take $e' = \langle \mathbf{e}' \rangle_i$. Thus, in the following, we can assume that $e \neq \langle \mathbf{e} \rangle_i$.

Case (context). Subcases $\mathbf{E} = [] \mid \mathbf{e_0}$ and $\mathbf{E} = \nu x : t.\ []$ are handled similarly to the proof of the corresponding lemma in [15]. Now we consider the remaining subcase $\mathbf{E} = (\mathsf{bind}\ x = [] \ \mathsf{in}\ \mathbf{e_0})$. Here $\mathbf{e} = (\mathsf{bind}\ x = \mathbf{e_1} \ \mathsf{in}\ \mathbf{e_0})$ and $\mathbf{e}' = (\mathsf{bind}\ x = \mathbf{e_1'} \ \mathsf{in}\ \mathbf{e_0})$. The premise of $\mathbf{e} \longrightarrow \mathbf{e}'$ gives $\mathbf{e_1} \longrightarrow \mathbf{e_1'}$. Then e is either $\langle \mathbf{e} \rangle_i$ or $\mathsf{bind}\ x = e_1 \ \mathsf{in}\ e_0$, where $\mathbf{e_1} = \pi_i e_1$ and $\mathbf{e_0} = \pi_i e_0$. The first case we already handled. In the second case, the induction hypothesis gives $e_1 \Longrightarrow e_1'$ for some e_1' such that $\mathbf{e_1'} \leq_0 \pi_i e_1'$. Then, by (context), $\mathsf{bind}\ x = e_1 \ \mathsf{in}\ e_0 \Longrightarrow \mathsf{bind}\ x = e_1' \ \mathsf{in}\ e_0$

holds, and $\mathbf{e}' = \mathbf{E}[\mathbf{e}'_1] \leq_0 \mathbf{E}[\pi_i e'_1] = \pi_i(\mathsf{bind}\ x = e'_1\ \mathsf{in}\ e_0)$. Thus we can take $e' = \mathsf{bind}\ x = e'_1\ \mathsf{in}\ e_0$.

Cases (app), (bind), (unwrap-subt), (unwrap-subt-else), (unwrap-pat), (unwrap-pat-else). Because $e \neq \langle \mathbf{e} \rangle_i$, the expression e can only contain brackets in its proper subexpressions (e.g. v, e_1, and e_2 for the (unwrap-subt) rule). Thus we can write e as $\hat{\mathbf{e}}[\overline{e_k}]$, where $\overline{e_k}$ are the proper subexpressions occurring in e, e.g. $e = \hat{\mathbf{e}}[v, e_1, e_2]$ for the (unwrap-subt) rule. Then $\mathbf{e} = \hat{\mathbf{e}}[\overline{\pi_i e_k}]$ because π_i is a homomorphism on $\hat{\mathbf{e}}$. We can also write $\mathbf{e}'$ as $\hat{\mathbf{e}}'[\overline{\pi_i e_k}]$. Because the reduction $\hat{\mathbf{e}}[\overline{\pi_i e_k}] \longrightarrow \hat{\mathbf{e}}'[\overline{\pi_i e_k}]$ holds and its premises and restrictions on the subexpressions (e.g. that the expression v must be a value) are invariant under projection (and inverse of projection), we can replace the subexpressions $\overline{\pi_i e_k}$ by $\overline{e_k}$ to get a derivation of the reduction $\hat{\mathbf{e}}[\overline{e_k}] \longrightarrow \hat{\mathbf{e}}'[\overline{e_k}]$. Take $e' = \hat{\mathbf{e}}'[\overline{e_k}]$. Then $e \longrightarrow e'$ and $\pi_i e' = \pi_i(\hat{\mathbf{e}}'[\overline{e_k}]) = \hat{\mathbf{e}}'[\overline{\pi_i e_k}] = \mathbf{e}'$ because π_i is a homomorphism on $\hat{\mathbf{e}}'$ (here we use Lemma 1 if necessary). □

3 Type System

The type system of the language is given in Figures 4 and 5. The types of channels and the corresponding subtyping rules are from [15]. As security levels we have only L and H, not an arbitrary lattice.

There are two forms of typing judgements. Both depend on the environment Γ, which consists of a list of typings of (value-level) variables and a list of type-level variables. The judgement form $\Gamma \vdash_{(pc)} N$ is used for some constructs related to communication. It asserts that the process N has the security level pc. Most judgements use the form $\Gamma \vdash e : \tau$, which asserts that e can be given the extended type τ. The two forms are related by the rule (P-NORMAL). The meta-expression tyvars(t) used in the rules denotes the set of type variables (of all three kinds) occurring in the type t.

We give types not only to expressions reducing to values but also to (expressions reducing to) procedures. This is different from [15] because we also need to be able to return procedures from functions. Procedures have an extended type of the form Proc_{pc}. This means that the procedure has the security level pc. If a procedure reads or writes to a channel then the security levels of the procedure and the channel must be equal. The rule (E-SUB) allows a procedure of type Proc_L (a low process) to have a subprocedure of type Proc_H (a high process) but not vice versa. Thus if a low process uses a high channel then the continuation (that follows the use of the channel) has type Proc_H. If it also needs to continue running in low context then the rule (P-PAR) allows it to fork into two low processes, one of which changes into high context before using the channel.

The type rules (E-VAR), (E-LAM), (E-APP), (E-BIND), and (E-SUB) are standard. The rules (N-SEND), (N-RECV), (N-NULL), (N-SUM), (P-NORMAL), (P-PAR), (P-REPL), (P-NEW), and (P-BRACKET) are similar to those in [15], except that we allow send arbitrary values ($\tilde{v}$) in (N-SEND), we have an explicit type annotation in (P-NEW), we have only one possible non-low security level in (P-BRACKET), and we rely on a separate rule (E-VAR) for typing variables.

$$t \le t \qquad \frac{}{l \le l} \qquad \frac{}{L \le H} \qquad \frac{}{p \le p} \qquad \frac{}{\pm \le +} \qquad \frac{}{\pm \le -}$$

$$\frac{t_2 \le t_1 \qquad \tau_1 \le \tau_2}{t_1 \to \tau_1 \le t_2 \to \tau_2} \qquad \frac{\forall i.\ t_i \le t_i'}{\tilde{t}_i \le \tilde{t}_i'} \qquad \frac{pc_2 \le pc_1}{\mathsf{Proc}_{pc_1} \le \mathsf{Proc}_{pc_2}}$$

$$\frac{p_1 \le p_2 \qquad (p_2 \le - \Rightarrow \tilde{t} \le \tilde{t}') \qquad (p_2 \le + \Rightarrow \tilde{t}' \le \tilde{t})}{\langle \tilde{t} \rangle_l^{p_1} \le \langle \tilde{t}' \rangle_l^{p_2}}$$

Figure 4. Subtyping rules

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t}\ \text{(E-VAR)} \qquad \frac{\Gamma \vdash v_1 : t_1 \to \tau_2 \qquad \Gamma \vdash v_2 : t_1}{\Gamma \vdash v_1\ v_2 : \tau_2}\ \text{(E-APP)}$$

$$\frac{\mathrm{tyvars}(t_1) \subseteq \Gamma \qquad \Gamma; f : t_1 \to \tau_2; x : t_1 \vdash e : \tau_2}{\Gamma \vdash (\mathsf{fix}\ f.\lambda x : t_1.\ e) : t_1 \to \tau_2}\ \text{(E-LAM)}$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma; x : t_1 \vdash e_2 : \tau_2}{\Gamma \vdash (\mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2) : \tau_2}\ \text{(E-BIND)} \qquad \frac{\Gamma \vdash v : t \qquad \mathrm{tyvars}(t) \subseteq \Gamma}{\Gamma \vdash (\mathsf{wrap}\ v : t) : \mathsf{Dynamic}}\ \text{(E-WRAP)}$$

$$\frac{\Gamma \vdash v : \mathsf{Dynamic} \qquad \Gamma; x : t \vdash e_1 : \tau' \qquad \Gamma \vdash e_2 : \tau' \qquad \mathrm{tyvars}(t) \subseteq \Gamma}{\Gamma \vdash (v\ \mathsf{unwrap}\ x :< t \succ e_1\ \mathsf{else}\ e_2) : \tau'}\ \text{(E-UNWRAP-SUBT)}$$

$$\frac{\Gamma \vdash v : \mathsf{Dynamic} \qquad \Gamma; x : t; \mathrm{tyvars}(t) \vdash e_1 : \tau' \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash (v\ \mathsf{unwrap}\ x : t \succ e_1\ \mathsf{else}\ e_2) : \tau'}\ \text{(E-UNWRAP-PAT)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \le \tau'}{\Gamma \vdash e : \tau'}\ \text{(E-SUB)} \qquad \frac{\Gamma \vdash e : \mathsf{Proc}_{pc}}{\Gamma \vdash\ !e : \mathsf{Proc}_{pc}}\ \text{(P-REPL)}$$

$$\frac{\Gamma \vdash x : \langle \tilde{t} \rangle_{pc}^{+} \qquad \Gamma \vdash \tilde{v} : \tilde{t} \qquad \Gamma \vdash e : \mathsf{Proc}_{pc}}{\Gamma \vdash_{(pc)} \bar{x}\,\langle \tilde{v} \rangle\ .\ e}\ \text{(N-SEND)}$$

$$\frac{\Gamma \vdash x : \langle \tilde{t} \rangle_{pc}^{-} \qquad \Gamma; \tilde{y} : \tilde{t} \vdash e : \mathsf{Proc}_{pc}}{\Gamma \vdash_{(pc)} x(\tilde{y}).\ e}\ \text{(N-RECV)}$$

$$\frac{}{\Gamma \vdash_{(pc)} \mathbf{0}}\ \text{(N-NULL)} \qquad \frac{\Gamma \vdash_{(pc)} M \qquad \Gamma \vdash_{(pc)} N}{\Gamma \vdash_{(pc)} M + N}\ \text{(N-SUM)}$$

$$\frac{\Gamma \vdash_{(pc)} N}{\Gamma \vdash N : \mathsf{Proc}_{pc}}\ \text{(P-NORMAL)} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Proc}_{pc} \qquad \Gamma \vdash e_2 : \mathsf{Proc}_{pc}}{\Gamma \vdash (e_1 \mid e_2) : \mathsf{Proc}_{pc}}\ \text{(P-PAR)}$$

$$\frac{\mathrm{tyvars}(t) \subseteq \Gamma \qquad \Gamma; x : t \vdash e : \mathsf{Proc}_{pc} \qquad t = \langle \tilde{t} \rangle_l^p}{\Gamma \vdash (\nu x : t.\ e) : \mathsf{Proc}_{pc}}\ \text{(P-NEW)}$$

$$\frac{\Gamma \vdash \mathbf{e} : \mathsf{Proc}_H}{\Gamma \vdash \langle \mathbf{e} \rangle_i : \mathsf{Proc}_H}\ \text{(P-BRACKET)}$$

Figure 5. The type system

We have $\mathrm{tyvars}(t) \subseteq \Gamma$ as a premise of some rules to ensure that only those type variables that are in scope are used.

The rule (E-WRAP) ensures that the value and the type wrapped in a dynamic correspond to each other. We also have two rules for the unwrap expressions. The rule (E-UNWRAP-SUBT) handles the variant that uses subtyping

to compare the dynamic and the static type. Here we require all type variables occurring in t to be in scope, because here we cannot bind new variables. The rule (E-UNWRAP-PAT) handles the variant that uses pattern matching. Here we add the variables tyvars(t) occurring in the pattern t to the scope. Currently, the rule considers all type variables in t to be new variables bound by the pattern even if there already was a variable with the same in the context. It would also be possible to allow some variables in the pattern to refer to the variables already in the scope if we syntactically distinguish these variables from pattern variables.

The two unwrap constructs cannot be united into one that allows both pattern matching and subtyping to be used. Let us consider the type $(\mathsf{Dynamic} \to \mathsf{Proc}_L) \to \mathsf{Proc}_H$ and the pattern $(\mathsf{Dynamic} \to \mathsf{Proc}_a) \to \mathsf{Proc}_a$. Then $(\mathsf{Dynamic} \to \mathsf{Proc}_L) \to \mathsf{Proc}_H \leq (\mathsf{Dynamic} \to \mathsf{Proc}_a) \to \mathsf{Proc}_a$ holds for both $a = L$ and $a = H$. Thus a is not uniquely defined and there is no reason to prefer either $a = L$ or $a = H$ because in both cases the inequality is strict (unlike for $\mathsf{Dynamic} \to \mathsf{Proc}_H \leq \mathsf{Dynamic} \to \mathsf{Proc}_a$, where also both $a = L$ and $a = H$ fit the inequality, but here we can choose $a = H$ because equality holds only in that case).

We will now prove some lemmas related to the type system. First, a lemma that allows substituting a variable with a value of the same type.

Lemma 4. *$\Gamma; y : t_1 \vdash e : \tau$ and $\Gamma \vdash v : t_1$ imply $\Gamma \vdash e[y \Leftarrow v] : \tau$.*

Proof. Take the derivation tree of $\Gamma; y : t_1 \vdash e : t$ and replace y with v. This replacement can invalidate only (E-VAR) nodes (used to derive $\Gamma \vdash y : t_1$). Now we have to derive $\Gamma \vdash v : t_1$ instead. Thus we replace these (E-VAR) nodes with the derivation tree of $\Gamma \vdash v : t_1$ which we have. Then we get a derivation tree of $\Gamma \vdash e[y \Leftarrow v] : \tau$. □

Next, we prove a similar lemma that allows substituting type variables in an expression with type-level entities (types, polarities, and security levels) of the same kind if we make the same substitution in the type of the expression.

Lemma 5. *If none of the variables $\overline{\alpha_j}, \overline{\beta_j}, \overline{\gamma_j}$ occur in Γ or τ then*
$\Gamma; y : t_1; \overline{\alpha_j}, \overline{\beta_j}, \overline{\gamma_j} \vdash e : \tau$ implies
$\Gamma; y : t_1[\overline{\alpha_j \Leftarrow \ell_j}, \overline{\beta_j \Leftarrow p_j}, \overline{\gamma_j \Leftarrow t_j}] \vdash e[\overline{\alpha_j \Leftarrow \ell_j}, \overline{\beta_j \Leftarrow p_j}, \overline{\gamma_j \Leftarrow t_j}] : \tau$.

Proof. Apply the substitution $[\overline{\alpha_j \Leftarrow \ell_j}, \overline{\beta_j \Leftarrow p_j}, \overline{\gamma_j \Leftarrow t_j}]$ to the derivation tree of $\Gamma; y : t_1; \overline{\alpha_j}, \overline{\beta_j}, \overline{\gamma_j} \vdash e : \tau$. The substitution does not change Γ (used in (E-VAR)). Also, the ordering of types (used in (E-SUB)) is not changed by the substitution. The statement tyvars(t) $\subseteq \Gamma$ (in (E-WRAP), (E-LAM), and (E-UNWRAP-SUBT)) is also not invalidated by the substitution. Elsewhere in the type rules, types or extended types that are denoted by a letter (e.g. t_1 or τ') and do not contain other such types, are used in a parametrically polymorphic way, thus the rules are not invalidated by the substitution. As a result of the substitution, after dropping the no longer used variables $\overline{\alpha_j}, \overline{\beta_j}, \overline{\gamma_j}$ from the context, we then get the derivation tree of $\Gamma; y : t_1[\overline{\alpha_j \Leftarrow \ell_j}, \overline{\beta_j \Leftarrow p_j}, \overline{\gamma_j \Leftarrow t_j}] \vdash e[\overline{\alpha_j \Leftarrow \ell_j}, \overline{\beta_j \Leftarrow p_j}, \overline{\gamma_j \Leftarrow t_j}] : \tau$. □

The following lemma allows substituting both value-level and type-level variables.

Lemma 6. *If none of the variables* $\overline{\alpha_j}, \overline{\beta_j}, \overline{\gamma_j}$ *occur in* Γ *or* τ *then* $\Gamma; y : t_1; \overline{\alpha_j}, \overline{\beta_j}, \overline{\gamma_j} \vdash e : \tau$ *and* $\Gamma \vdash v : t_1[\overline{\alpha_j \Leftarrow \ell_j}, \overline{\beta_j \Leftarrow p_j}, \overline{\gamma_j \Leftarrow t_j}]$ *imply* $\Gamma \vdash e[x \Leftarrow v, \overline{\alpha_j \Leftarrow \ell_j}, \overline{\beta_j \Leftarrow p_j}, \overline{\gamma_j \Leftarrow t_j}] : \tau$.

Proof. Combine Lemmas 4 and 5. □

Now we can prove Subject Reduction, which is the main lemma needed for non-interference. It shows that the security type of an expression does not change during reduction. Thus, if we have two bisimilar expressions of equal extended types, they will continue to have equal extended types when we simulate the reduction steps of one of the expressions in the other.

Lemma 7 (Subject Reduction). *If* $e \longrightarrow e'$ *then* $\Gamma \vdash e : \tau$ *implies* $\Gamma \vdash e' : \tau$.

Proof. By induction on the derivation of $e \longrightarrow e'$. We can assume that (E-SUB) is never used immediately above another (E-SUB) because two or more successive instances of (E-SUB) can always be replaced by one. We can also assume that (E-SUB) is not the bottommost rule in the derivation tree of $e \longrightarrow e'$.

Case (bind). $\Gamma \vdash (\mathsf{bind}\ x = v\ \mathsf{in}\ e_2) : \tau$. The bottommost rule must be (E-BIND), whose premises give $\Gamma \vdash v : t_1$ and $\Gamma; x : t_1 \vdash e_2 : \tau$. Lemma 4 now gives $\Gamma \vdash e_2[x \Leftarrow v] : \tau$.

Case (app). $\Gamma \vdash (\mathsf{fix}\ f.\lambda x : t_1.\ e_2)\ v : \tau$. By (E-APP), $\Gamma \vdash (\mathsf{fix}\ f.\lambda x : t_1.\ e_2) : t_1 \to \tau$. By (E-SUB) and (E-LAM), $\Gamma; f : t_1 \to \tau_2; x : t_1 \vdash e_2 : \tau_2$ and $\Gamma \vdash v : t_1$, where $\tau_2 \leq \tau$. Lemma 4 and (E-SUB) now give $\Gamma \vdash e_2[f \Leftarrow (\mathsf{fix}\ f.\lambda x : t_1.\ e_2), x \Leftarrow v] : \tau$.

Case (unwrap-subt). $\Gamma \vdash ((\mathsf{wrap}\ v : t_1)\ \mathsf{unwrap}\ x :< t_2 \succ e_1\ \mathsf{else}\ e_2) : \tau$. By (E-UNWRAP-SUBT) and (E-WRAP) (we can assume that (E-SUB) is not used below (E-WRAP) because Dynamic is not a supertype of anything but itself), $\Gamma \vdash v : t_1$. (E-SUB) and the premise of (unwrap-subt) give $\Gamma \vdash v : t_2$. Combining this with another premise of (E-UNWRAP-SUBT) and Lemma 4, gives $\Gamma \vdash e_1[x \Leftarrow v] : \tau$.

Case (unwrap-pat). $\Gamma \vdash ((\mathsf{wrap}\ v : t_1)\ \mathsf{unwrap}\ x : t_2 \succ e_1\ \mathsf{else}\ e_2) : \tau$. By (E-UNWRAP-PAT), (E-SUB), and (E-WRAP), $\Gamma \vdash v : t_1$ and $\Gamma; x : t_2; \mathrm{tyvars}(t_2) \vdash e_1 : \tau$. The premise of (unwrap-pat) gives $\Gamma \vdash v : t_2[\overline{\alpha_j \Leftarrow \ell_j}, \overline{\beta_j \Leftarrow p_j}, \overline{\gamma_j \Leftarrow t_j}]$. Lemma 6 (we can use alpha-conversion, if necessary, to achieve that the variables $\mathrm{tyvars}(t_2)$ do not occur in Γ or τ, as required by the lemma) now gives $\Gamma \vdash e_1[x \Leftarrow v, \overline{\alpha_j \Leftarrow \ell_j}, \overline{\beta_j \Leftarrow p_j}, \overline{\gamma_j \Leftarrow t_j}] : \tau$.

Cases (unwrap-subt-else) and (unwrap-pat-else). Immediate (an expression reduces to a subexpression of the same type).

Case (context). $\Gamma \vdash E[e_1] : \tau$. The premise of (context) and the induction hypothesis give $e_1 \longrightarrow e_1'$, where $\Gamma' \vdash e_1 : \tau_1$, $\Gamma' \vdash e_1' : \tau_1$, and $e' = E[e_1']$. For (E-BIND), (P-PAR), and (P-BRACKET), $\Gamma' = \Gamma$. For (P-NEW), $\Gamma' = \Gamma; x : t$, where $E = \nu x : t.\ []$. In each case we can replace e_1 with e_1' (and the premise

$\Gamma' \vdash e_1 : \tau_1$ with $\Gamma' \vdash e_1' : \tau_1$) in the derivation tree of $\Gamma \vdash E[e_1] : \tau$ to get $\Gamma \vdash E[e_1'] : \tau$.

Cases (comm), (extr), (new), (repl), (split), (glue), (break), and (push) are handled similarly to the proof of the corresponding lemma in [15]. □

4 Non-Interference

The non-interference results for our language are similar to those in [15] and we omit the proofs here. The proofs use Lemmas 2, 3, and 7, which were proved for our language in the earlier sections. The necessary definitions are almost the same as in [15]:

Definition 5. *Let α,β,... range over names and co-names (x,$\bar{x}$,...). If α is x or $\bar{x}$ then $|\alpha|$ is x. The predicate $e \downarrow_\alpha$ (read: the expression e is observable at α) is defined as follows:*

$$(N + x(\tilde{y}).\ e) \downarrow_x \qquad (N + \bar{x}\,\langle \tilde{y} \rangle\,.\ e) \downarrow_{\bar{x}} \qquad \frac{e \downarrow_\alpha \qquad E \text{ does not bind } |\alpha|}{E[e] \downarrow_\alpha}$$

The evaluation context here is

$$E ::= ([] \mid e') \mid (e' \mid []) \mid \nu x : t.\ [] \mid \langle [] \rangle_i$$

i.e. it does not include bind *expressions.*

$e \Downarrow_\alpha$ *stands for* $(\exists e'.\ e \Longrightarrow e' \downarrow_\alpha)$.

Definition 6. *Let B be an arbitrary set of names. A binary relation $\mathcal{R}$ over processes is a* weak B-simulation *if and only if*

- $e_1\ \mathcal{R}\ e_2 \wedge e_1 \Longrightarrow e_1'$ *implies* $\exists e_2'.\ e_2 \Longrightarrow e_2' \wedge e_1'\ \mathcal{R}\ e_2'$ *and*
- $|\alpha| \in B$, $e_1\ \mathcal{R}\ e_2$, *and* $e_1 \Downarrow_\alpha$ *imply* $e_2 \Downarrow_\alpha$.

$\mathcal{R}$ is a weak B-bisimulation *if and only if $\mathcal{R}$ and $\mathcal{R}^{-1}$ are weak B-simulations.*

Definition 7. *Given a type environment Γ,* $\mathrm{low}(\Gamma)$ *denotes the largest set $B \subseteq \mathcal{N}$ ($\mathcal{N}$ is the set of all names) such that $x \in B$ and $\Gamma(x) = \langle \tilde{t} \rangle_l^p$ imply $l = L$.*

Lemma 8. *Let $\mathbf{e_1}\ \mathcal{R}_\Gamma\ \mathbf{e_2}$ hold if and only if, for some process e and some extended type τ, both $\Gamma \vdash e : \tau$ and $\mathbf{e_1} \leq_0^* \cdot\ \pi_1^{-1}\ e\ \pi_2 \cdot \geq_0^* \mathbf{e_2}$ hold. Then, $\mathcal{R}_\Gamma$ is a weak* $\mathrm{low}(\Gamma)$*-bisimulation.*

Theorem 1. *Assume that $\Gamma_0 \vdash \mathbf{e}_i : \mathsf{Proc}_H$ holds for $i \in \{1, 2\}$. Then, for any context $\mathbf{C}$ and for any environment Γ such that $\Gamma \vdash \mathbf{C}[]$ holds under the assumption $\Gamma_0 \vdash [] : \mathsf{Proc}_H$, $\mathbf{C}[\mathbf{e}_1]$ and $\mathbf{C}[\mathbf{e}_2]$ are weakly* $\mathrm{low}(\Gamma)$*-bisimilar.*

Thus, replacing one high subprocedure (subexpression of type Proc_H) with another in an expression changes neither the set of low channels that it can read nor the set of low channels that it can write to.

5 More Examples

Our language does not support global definitions, but in the following, we allow them as syntactic sugar. Thus, if we have global definitions $x_1 = e_1, \ldots, x_n = e_n$ and a main expression e then we assume the full program to be

$$\mathsf{bind}\ x_1 = e_1\ \mathsf{in}\ \ldots \mathsf{bind}\ x_n = e_n\ \mathsf{in}\ e$$

Here is some other syntactic sugar that can be used by the programmer:

$$\lambda x.\ e \equiv \mathsf{fix}\ f.\lambda x.\ e \qquad \text{where } f \text{ is not free in } e$$

Because our language does not have a unit type, we define a dummy value of type Dynamic:

$$dummyDyn = \mathsf{wrap}\ (\lambda x : \mathsf{Dynamic}.\ x) : \mathsf{Dynamic} \to \mathsf{Dynamic}$$

Our language allows using dynamics with pattern matching, similarly to the dynamics in Clean [19]. For example, we can define a function for dynamic function applicaton. For ordinary functions this is easy but we can also define such a function for functions that return procedures. Procedures cannot directly return a value but we can simulate this by using a channel where the procedure will write its result. We can create a fresh channel for each procedure call, so that it would not interfere with other channels. This is done in

$$\begin{array}{l}
dynAppProc = \\
\quad \lambda dynF : \mathsf{Dynamic}.\ \lambda dynCX : \mathsf{Dynamic}.\ \lambda cont : \mathsf{Dynamic} \to \mathsf{Proc}_L. \\
\quad\quad dynF\ \mathsf{unwrap}\ f : \gamma_1 \to \langle \gamma_2 \rangle^{+}_{\alpha} \to \mathsf{Proc}_\alpha \succ \\
\quad\quad\quad dynCX\ \mathsf{unwrap}\ cX :< \langle \gamma_1 \rangle^{\pm}_{\alpha} \succ \nu cRes : \langle \gamma_2 \rangle^{\pm}_{\alpha}\,. \\
\quad\quad\quad\quad\quad (cont\ (\mathsf{wrap}\ cRes : \langle \gamma_2 \rangle^{\pm}_{\alpha}) \mid cX(x).\ (\overline{cX}\,\langle x \rangle\,.\ \mathbf{0} \mid f\ x\ cRes)) \\
\quad\quad\quad\quad \mathsf{else}\ dummyDyn \\
\quad\quad\quad \mathsf{else}\ dummyDyn
\end{array}$$

The channel where the result will appear is given to the continuation *cont* because nothing can be returned from a procedure to the original caller. The continuation gets the channel immediately, not after f terminates. The value will appear on the channel later. This allows the continuation to run in low context.

The channel $cRes$ can be used (only in high context, if $\alpha = H$) to wait for f to return a value and to read the value, thus $cRes$ is essentially a *future* [7]. But $cRes$ can be given as an argument to another high procedure (let it be g). Then g can start running only after a value has arrived at $cRes$ but the future of g is still returned immediately, even before the value has arrived at $cRes$. This allows to sequence high procedures while remaining in low context. This is done here:

$$\begin{array}{l}
dynAppProc\ (\mathsf{wrap}\ f)\ arg\ (\lambda cRes : \mathsf{Dynamic}. \\
\quad dynAppProc\ (\mathsf{wrap}\ g)\ cRes\ (\lambda futG : \mathsf{Dynamic}.\ \overline{clow}\,\langle futG \rangle\,.\ \mathbf{0}))
\end{array}$$

(here f and g are functions of type $\mathsf{Dynamic} \to \langle \mathsf{Dynamic} \rangle_H^+ \to \mathsf{Proc}_H$ and $clow$ is a channel of type $\langle \mathsf{Dynamic} \rangle_L^\pm$; we omit the annotations in wrap construct, as they can be inferred by the type checker). The value $futG$ can appear at $clow$ before f and g have terminated.

Only when waiting for the termination of or reading the result of a high procedure is desired, is it necessary to switch into high context.

$clow(futG)$.

$$futG \ \mathsf{unwrap} \ chigh : \langle \mathsf{Dynamic} \rangle_H^\pm \succ chigh(resG).\ handle\ resG \ \mathsf{else}\ \mathbf{0}$$

Here $handle\ resG$ must have type Proc_H and it cannot be executed before f and g have terminated and g has returned the result $resG$.

This is why we do not distinguish high and low values (of types like $\mathsf{Dynamic}_H$ and $\mathsf{Dynamic}_L$) in our language but only high and low channels. The level of a value that is not on a channel is the level of the context where it is handled. If we need to use a high value in a low context, we can create a high channel and use it as a future of the high value. This was done with $cRes$ and $futG$ above.

6 Related Work

Program analyses for secure information flow were first considered by Denning [9], the type systems stem from the work of Volpano et al. [20]. The non-interference property was first proposed in its modern form by Goguen and Meseguer [10], while the testing equivalence that we use to express it stems from [8,6].

A general type discipline for information flow security in π-calculus was proposed by Honda et al. [12]. Our work, however, mostly follows Pottier [15] and adopts the $\langle\pi\rangle$-calculus presented there for arguing about two processes simultaneously. Similar compositions and argument systems have also been considered for secure information flow in imperative and object-oriented languages [5,18].

Our work also extends the existing type-based information flow analyses for higher-order languages. The SLam calculus [11], DCC [1] and FlowCaml [16] are some existing higher-order calculi allowing reasoning about information flow.

Recently, there has been interest in dynamic enforcement of information flow policies [4,17]. These mechanisms can handle more lax control structures of programs, similarly to the dynamic types in this paper. Our enforcement mechanism is fully static but it would be interesting to compare it to dynamic mechanisms.

7 Conclusions

We have presented a type system for information flow analysis for a π-calculus extended with recursive λ-abstractions and dynamics. Such a language can be used to model distributed systems where procedure code of different security levels can be sent over channels and some of the channels must ensure secrecy. We saw that the added constructs in our language do not weaken the non-interference guarantees compared to the ordinary π-calculus.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In Appel and Aiken [3], pages 147–160.
2. M. Abadi, L. Cardelli, B. C. Pierce, and G. D. Plotkin. Dynamic Typing in a Statically-Typed Language. In *POPL*, pages 213–227. ACM Press, 1989.
3. A. W. Appel and A. Aiken, editors. *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. ACM, 1999.
4. A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *CSF*, pages 43–59. IEEE Computer Society, 2009.
5. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSFW*, pages 100–114. IEEE Computer Society, 2004.
6. M. Boreale and R. De Nicola. Testing equivalence for mobile processes. *Inf. Comput.*, 120(2):279–303, 1995.
7. F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In R. D. Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2007.
8. R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theor. Comput. Sci.*, 34:83–133, 1984.
9. D. E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM*, 19(5):236–243, 1976.
10. J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
11. N. Heintze and J. G. Riecke. The slam calculus: Programming with secrecy and integrity. In D. B. MacQueen and L. Cardelli, editors, *POPL*, pages 365–377. ACM, 1998.
12. K. Honda, V. T. Vasconcelos, and N. Yoshida. Secure Information Flow as Typed Process Behaviour. In G. Smolka, editor, *ESOP*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199. Springer, 2000.
13. A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In Appel and Aiken [3], pages 228–241.
14. J. Parrow. An Introduction to the π-Calculus. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 479–543. Elsevier, 2001.
15. F. Pottier. A Simple View of Type-Secure Information Flow in the π-Calculus. In *CSFW*, pages 320–330. IEEE Computer Society, 2002.
16. F. Pottier and V. Simonet. Information Flow Inference for ML. In J. Launchbury and J. C. Mitchell, editors, *POPL*, pages 319–330. ACM, 2002.
17. A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *CSF*, pages 186–199. IEEE Computer Society, 2010.
18. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In C. Hankin and I. Siveroni, editors, *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2005.
19. T. van Noort, P. Achten, and R. Plasmeijer. Ad-hoc Polymorphism and Dynamic Typing in a Statically Typed Functional Language. In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, WGP '10, pages 73–84, New York, NY, USA, 2010. ACM.
20. D. M. Volpano, C. E. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.