Secure Composition of Untrusted Code: Wrappers and Causality Types

Peter Sewell
Computer Laboratory
University of Cambridge
Peter.Sewell@cl.cam.ac.uk

Jan Vitek
Department of Computer Sciences
Purdue University
jv@cs.purdue.edu

Abstract

We consider the problem of assembling concurrent software systems from untrusted or partially trusted off-the-shelf components, using *wrapper programs* to encapsulate components and enforce security policies. In previous work we introduced the *box-π* process calculus with constrained interaction to express wrappers and discussed the rigorous formulation of their security properties. This paper addresses the verification of wrapper information flow properties. We present a novel *causal type system* that statically captures the allowed flows between wrapped possibly-badly-typed components; we use it to prove that an example unidirectional-flow wrapper enforces a causal flow property.

1 Introduction

A typical desktop software environment nowadays contains components – whole programs, plug-ins, or smaller code fragments – obtained from different untrusted or partially-trusted sources; they interact in intricate ways. Components may be faulty or malicious, or designed with a weaker security policy that the user requires – what is legitimate marketing data to a vendor may be considered sensitive by a user. It is difficult for a user to gain assurance that the composed system is secure, particularly because many off-the-shelf components are only available as object code. Furthermore current operating systems fail to provide support for the kind of fine-grained policies that could control the execution of such components [GWTB96, FBF99].

Recent practical work advocates interposing security code at the operating system boundary to observe and modify the data passing through [WBDF97, Jon99, GRPA97, GWTB96, FBF99]. Interposition techniques effectively encapsulate untrusted components in *wrapper programs* that have full control over the interactions between encapsulated components and the OS and over the interactions among components. The code of a wrapper can, for instance, perform access control checks, audit, attempt to detect intruders, and even monitor covert channels. In [FBF99] Fraser, Badger and Feldman presented a system that splits the task of writing a wrapper into two parts. The wrapper's body is written in a variant of C called the Wrapper Definition Language. The dynamic aspects of creating wrappers and instantiating concurrently executing components are specified in the Wrapper Life Cycle framework. While quite expressive, their approach does not provide guarantees that the wrappers actually enforce the desired security policies. The powerful wrapper language, the fact that all wrappers execute in kernel mode, and the fact that components are concurrent combine to make it difficult to understand precisely what properties a wrapper enforces.

Our work is exploring secure composition using wrappers, focussing on the rigorous statement and proof of their security properties. To this end, we have abstracted the essential characteristics of the problem in a process calculus – powerful enough to express the code of non-trivial wrappers and to express the concurrent composition of components, but small enough to be amenable to formal proof. In this paper we study information flow properties of wrappers. To express clear statements of such properties we equip our calculus with an annotated operational semantics, regarding a wrapper and each wrapped component as a different principal and colouring processes with the sets of principals that have causally affected them. This allows a direct statement of the property that one component cannot causally affect another. Verifying such a causal flow property directly can be laborious, requiring a characterisation of the state space of a wrapper containing arbitrary components. We therefore introduce a type system that statically captures causal flows. Since components are often provided as object code, which is impractical for the user to typecheck, our type system must admit programs with badly-typed subcomponents.

Expressing wrappers requires a language for composing concurrently-executing components, including primitives for encapsulating components and controlling their interactions. We use the *box-π* calculus of [SV99a], recapitulated in Sections 2 and 3. Box-π is a minimal extension of the π-calculus with encapsulation; it is sufficiently expressive for components and wrappers while retaining the simplicity and tractable semantics needed for proving properties. Moreover Pict [PT99] demonstrates how to build a real programming language above a π-calculus core, a similar approach could be used for box-π.

Our main example, in Section 4, is a unidirectional-flow wrapper that encapsulates two components, allowing messages to be sent only in one direction between them and both components to interact with the environment. The following box-π program is a simplified version of this example.

$$(\boldsymbol{\nu}\, a, b)(\quad a[P] \quad | \quad !\, c^a x.\overline{c}^b x \quad | \quad b[Q] \quad)$$

Processes P and Q are arbitrary, possibly malicious, components. They are encapsulated in named boxes, with private names a and b, and placed in parallel with a forwarder process on channel c from box a to box b. The term $\overline{c}^b x$ is an output to channel c in box b of value x. The term $c^a x.\overline{c}^b x$ prefixes this with an input on channel c from box a; here the first x is a formal parameter that binds the second. The **!** operator indicates a replicated input, so the forwarder persists after use. The boxes restrict communication of the encapsulated processes and ensure that P and Q cannot interact with each other directly; the private names ensure that they cannot interact with their environment in any other way. This simplified forwarder sends only unordered asynchronous messages; our main example provides FIFO communication (this is related to the NRL pump [KML96], as discussed in Section 4).

Intuitively the system enforces an information flow policy that prevents Q from leaking secrets to P. When one attempts to make such properties precise, however, there are many choices. A body of model-theoretic work on *non-interference* uses delicate extensional properties of the trace sets of systems. In our programming language setting a more intensional approach allows what we believe to be clearer statements. We start with a labelled transition semantics that specifies the input/output behaviour of programs and extend it to represent and propagate causal dependencies explicitly. In terms of this, one can state the desired property as 'no visible action of P is causally dependent on any action of Q'. The causal semantics and property are defined in Section 5.

The causal type system, given in Section 6, allows us to prove information flow properties of box-π programs. For the example above, to statically allow the flow from a to b but disallow the converse we can associate the components with principals p and q, then take a to be a box name whose contents may be affected by p, written $a : \mathbf{box}_{\{\mathsf{p}\}}$, b to be a box name whose contents may be affected by p or q, written $b : \mathbf{box}_{\{\mathsf{p},\mathsf{q}\}}$, and c to be a channel, carrying values of a top type $\top$, which can be affected only by p, so $c : \mathbf{chan}_{\{\mathsf{p}\}} \top$. The fragment is then typable, whereas the converse forwarder $c^b x.\overline{c}^a x$ is not. The type system also deals with tracking causes through computation within a wrapper, including communication of channel names, and with interaction between a wrapper and badly-typed components. All boxes are assumed to contain untyped processes; wrapper code is statically typed; run-time type checking is required only when receiving from a component.

Further discussion of related work is given in Section 7; Section 8 concludes with future work. Proofs can be found in the technical report [SV99c].

2 A Boxed π Calculus

The language – known as the *box-π calculus* – that we use for studying encapsulation properties must allow interacting components to be composed. The components will typically be executing concurrently, introducing nondeterminism. It is therefore natural to base the language on a process calculus. The box-π calculus lies in a large design space of distributed calculi that build on the π-calculus of Milner, Parrow and Walker [MPW92], including among others the related calculi [AFG98, CG98, FGL+96, RH98, Sew98, SWP99, VC98]. A brief overview of the design space can be found in [Sew99]; here we highlight the main design choices for box-π.

The calculus is based on asynchronous message passing, with components interacting only by the exchange of unordered asynchronous messages. Box-π has an asynchronous π-calculus as a subcalculus – we build on a large body of work studying such calculi, notably [HT91, Bou92]. They are known to be very expressive, supporting many programming idioms including functions and objects, and are Turing-complete; a box-π process may therefore perform arbitrary internal computation. The choice of asynchronous communication is important as it allows two components to interact without creating causal connections in both directions between them.

Box-π requires facilities for constraining communication – in standard π-calculi, if one process can send a message to another then the only way to prevent information flowing in the reverse direction is to impose a type system on components, which (as observed above) is not appropriate here. We therefore add a boxing primitive – boxes may be nested, giving hierarchical protection domains; communication across box boundaries is strictly limited. Underlying the calculus design is the principle that each box should be able to control all interactions of its children, both with the outside world and with each other. Boxes can be viewed as protection domains, akin to operating system-enforced address spaces. Direct communication is therefore allowed only between a box and its parent, or within the process running in a particular box. All other communication, in particular that between two sibling boxes, must be mediated by code running in the parent. This code can enforce an arbitrary security policy, even supporting dynamically-changing policies and interfaces (in contrast to static restriction or blocking operators [BHR84, VD98]).

Turning to the values that may be communicated, it is convenient to allow arbitrary tuples of names (or other tuples). Note that we do *not* allow communication of process terms. Moreover, no primitives for movement of boxes are provided, in contrast to most work cited above. The calculus is therefore entirely first order, which is important for the tractable theory of behaviour (the labelled transition semantics) that we require to state and prove security properties. The calculus is also untyped – we wish to consider the wrapping of ill-understood, probably buggy and possibly malicious programs.

2.1 Syntax

The syntax of the calculus is as follows:

Names We take an infinite set $\mathcal{N}$ of *names*, ranged over by a, b, c etc. (except i, j, k, o, p, u, v). Both boxes and communication channels are named; names also play the role of variables, as in the π-calculus.

Values and Patterns Processes will interact by communicating values which are deconstructed by pattern-matching by the receiver. Values u, v can be names or tuples, with patterns p correspondingly tuple-structured.

u, v	$::=$	x	name
		$\langle v_1 \mathinner{..} v_k \rangle$	tuple ($k \geq 0$)

p	$::=$	$\_$	wildcard
		x	name pattern
		$(p_1 \mathinner{..} p_k)$	tuple pattern
			($k \geq 0$, no repeated names)

Processes The main syntactic category is that of *processes*, ranged over by P, Q. We introduce the primitives in three groups.

Boxes A box $n[P]$ has a name n, it can contain an arbitrary process P. Box names are not necessarily unique – the process $n[0] \mid n[0]$ consists of two distinct boxes named n, both containing an empty process, in parallel.

P	$::=$	$n[P]$	box named n containing P
		$P \mid P'$	P and P' in parallel
		0	the nil process

Communication The standard asynchronous π-calculus communication primitives are $\overline{x}v$, indicating an output of value v on the channel named x, and $xp.P$, a process that will receive a value output on channel x, binding it to p in P. Here we refine these with a tag indicating the direction of the communication in the box hierarchy. An *input tag* ι can be either $\star$, for input within a box, $\uparrow$, for input from the parent box, or a name n, for input from a sub-box named n. An *output tag* o can be any of these, similarly. For technical reasons we must also allow an output tag to be $\overline{\uparrow}$, indicating an output received from the parent that has not yet interacted with an input, or $\overline{n}$, indicating an output received from child n that has not yet interacted. The communication primitives are then

P	$::=$	$\ldots$	
		$\overline{x}^o v$	output v on channel x to o
		$x^\iota p.P$	input on channel x from ι
		$!\, x^\iota p.P$	replicated input

The replicated input $!\, x^\iota p.P$ behaves essentially as infinitely many copies of $x^\iota p.P$ in parallel. This gives computational power, allowing e.g. recursion to be encoded simply, while keeping the theory simple. In $x^\iota p.P$ and $!\, x^\iota p.P$ the names occurring in the pattern p bind in P. Empty patterns and tuples will often be elided.

New name creation Both box and channel names can be created fresh, with the standard π-calculus $(\boldsymbol{\nu}\, x)P$ operator. This declares any free instances of x within P to be instances of a globally fresh name.

P	$::=$	$\ldots$	
		$(\boldsymbol{\nu}\, x)P$	new name creation

In $(\boldsymbol{\nu}\, x)P$ the x binds in P. We work up to alpha conversion of bound names throughout, writing the free name function, defined in the obvious way for values, tags and processes, as $\mathrm{fn}(\_)$.

2.2 Semantics

This subsection defines the operational semantics of Box-π. The reader unfamiliar with process calculi may wish to skim to the start of Section 3 on a first reading.

2.2.1 Reduction Semantics

The simplest semantic definition of the calculus is a *reduction semantics*, a one-step reduction relation $P \rightarrow P'$ indicating that P can perform one step of internal computation to become P'. We first define the complement $\overline{\iota}$ of a tag ι in the obvious way, with $\overline{\star} = \star$ and $\overline{\overline{\iota}} = \iota$. We define a partial function $\{\_/\_\}$, taking a pattern and a value and giving, where it is defined, a partial function from names to values.

$$\begin{array}{rcl} \{{}^{v}/_{\_}\} & = & \{\} \\ \{{}^{v}/_{x}\} & = & \{x \mapsto v\} \\ \{{}^{\langle v_1 \,..\, v_{k'}\rangle}/_{\langle p_1 \,..\, p_k\rangle}\} & = & \{{}^{v_1}/_{p_1}\} \cup \ldots \cup \{{}^{v_k}/_{p_k}\} \text{ if } k = k' \\ & & \text{undefined, otherwise} \end{array}$$

The natural definition of the application of a substitution σ (from names to values) to a process term P, written σP, is also a partial operation, as the syntax does not allow arbitrary values in all the places where free names can occur. We write $\{{}^{v}/_{p}\}P$ for the result of applying the substitution $\{{}^{v}/_{p}\}$ to P. This may be undefined either because $\{{}^{v}/_{p}\}$ is undefined, or because $\{{}^{v}/_{p}\}$ is a substitution but the application of that substitution to P is undefined. For example, $\{{}^{\langle z\, z\rangle}/_{x}\}\overline{x}^{\star}\langle\rangle$ is not defined as $\overline{\langle z\, z\rangle}^{\star}\langle\rangle$ is not in the syntax. Note that the result $\{{}^{y}/_{x}\}P$ of applying a name-for-name substitution is always defined. This definition of substitution leads to a lightweight notion of runtime error[1].

The definition of reduction involves an auxiliary *structural congruence* $\equiv$, defined as the least congruence relation such that the axioms below hold. This allows the parts of a redex to be brought syntactically adjacent.

$$\begin{array}{rcll} P \mid Q & \equiv & Q \mid P & \\ (P \mid Q) \mid R & \equiv & P \mid (Q \mid R) & \\ (\boldsymbol{\nu}\, x)(\boldsymbol{\nu}\, y)P & \equiv & (\boldsymbol{\nu}\, y)(\boldsymbol{\nu}\, x)P & \\ (\boldsymbol{\nu}\, x)(P \mid Q) & \equiv & P \mid (\boldsymbol{\nu}\, x)Q & x \notin \mathrm{fn}(P) \\ (\boldsymbol{\nu}\, x)n[P] & \equiv & n[(\boldsymbol{\nu}\, x)P] & x \neq n \end{array}$$

The reduction relation is now the least relation over processes satisfying the axioms and rules below. The (Red Comm) and (Red Repl) axioms are subject to the condition that $\{{}^{v}/_{p}\}P$ is well-defined.

$$\begin{array}{ll} n[\overline{x}^{\uparrow}v \mid Q] \to \overline{x}^{\overline{n}}v \mid n[Q] & \text{(Red Up)} \\ \overline{x}^{n}v \mid n[Q] \to n[\overline{x}^{\overline{\uparrow}}v \mid Q] & \text{(Red Down)} \\ \overline{x}^{\bar{\iota}}v \mid x^{\iota}p.P \to \{{}^{v}/_{p}\}P & \text{(Red Comm)} \\ \overline{x}^{\bar{\iota}}v \mid\, !\, x^{\iota}p.P \to\, !\, x^{\iota}p.P \mid \{{}^{v}/_{p}\}P & \text{(Red Repl)} \\ P \to Q \;\;\Rightarrow\;\; P \mid R \to Q \mid R & \text{(Red Par)} \\ P \to Q \;\;\Rightarrow\;\; (\boldsymbol{\nu}\, x)P \to (\boldsymbol{\nu}\, x)Q & \text{(Red Res)} \\ P \to Q \;\;\Rightarrow\;\; n[P] \to n[Q] & \text{(Red Box)} \\ P \equiv P' \to Q' \equiv Q \;\;\Rightarrow\;\; P \to Q & \text{(Red Struct)} \end{array}$$

The (Red Up) axiom allows an output to the parent of a box to cross the enclosing box boundary. Similarly, the (Red Down) axiom allows an output to a child box n to cross the boundary of n. The (Red Comm) axiom then allows synchronisation between a complementary output and input within the same box. The (Red Repl) axiom is similar, but preserves the replicated input in the resulting state.

Communications across box boundaries take two reduction steps, as in the following upwards and downwards communications.

$$\begin{array}{rcl} n[\overline{x}^{\uparrow}v] \mid x^{n}p.P & \to & n[0] \mid \overline{x}^{\overline{n}}v \mid x^{n}p.P \\ & \to & n[0] \mid \{{}^{v}/_{p}\}P \end{array}$$

$$\begin{array}{rcl} \overline{x}^{n}v \mid n[x^{\uparrow}p.P] & \to & n[\overline{x}^{\overline{\uparrow}}v \mid x^{\uparrow}p.P] \\ & \to & n[\{{}^{v}/_{p}\}P] \end{array}$$

This removes the need for 3-way synchronisations between a box, an output and an input (as in [VC98]), simplifying both the semantics and the implementation model.

2.2.2 Labelled Transitions

The reduction semantics defines only the internal computation of processes. The statements of our security properties must involve the interactions of processes with their environments, requiring more structure: a *labelled transition relation* characterising the potential inputs and outputs of a process. We give a labelled semantics for box-π in an explicitly-indexed early style, defined inductively on process structure by a structured operational semantics. The *labels* are

$$\begin{array}{rcll} \ell & ::= & \tau & \text{internal action} \\ & & \overline{x}^{o}v & \text{output action} \\ & & x^{\gamma}v & \text{input action} \end{array}$$

where γ ranges over tags $\star$, n, $\uparrow$ and $\overline{n}$. The labelled transitions can be divided into those involved in moving messages across box boundaries and those involved in communications between outputs and inputs. The movement labels are

$\overline{x}^{n}v$ (sending to child n)
$x^{\overline{n}}v$ (box n receiving from its parent)
$\overline{x}^{\uparrow}v$ (sending to the parent)

Say $\mathrm{mv}(o)$ is true if o is of the form n or $\uparrow$. The communication labels are

$\overline{x}^{\star}v$ (local output)
$x^{\star}v$ (local input)
$\overline{x}^{\overline{n}}v$ (output received from child n)
$x^{n}v$ (input a message received from child n)
$\overline{x}^{\overline{\uparrow}}v$ (output received from parent)
$x^{\uparrow}v$ (input a message received from parent)

Labels synchronise in the pairs $\overline{x}^{\bar{\gamma}}v$ and $x^{\gamma}v$. The labelled transition relation has the form

$$A \vdash P \xrightarrow{\ell} Q$$

[1] A more conventional notion of runtime error would give errors only when a tuple is used as a name, e.g. for output. The substitution-based notion is forced by our choice of syntax, which disallows values in various places where names may appear. In general it will report errors sooner than the conventional notion.

where A is a finite set of names and $\mathrm{fn}(P) \subseteq A$; it should be read as 'in a state where the names A may be known to P and its environment, process P can do ℓ to become Q'. The relation is defined as the smallest relation satisfying the rules in Figure 3 omitting all transition subscripts, occurrences of C : and occurrences of $\mathsf{C}\bullet$. We write A, x for $A \cup \{x\}$ where x is assumed not to be in A, and A, p for the union of A and the names occurring in the pattern p, where these are assumed disjoint.

The labelled semantics is explained further in [SV99a]. It is similar to a standard π semantics but must also deal with boxes and with reductions such as

$$((\boldsymbol{\nu}\, x)\overline{x}^n z) \mid n[0] \quad \rightarrow \quad (\boldsymbol{\nu}\, x)n[\overline{x}^{\uparrow} z]$$

in which a new-bound name enters a box boundary.

The two semantics coincide in the following sense.

Theorem 1 *If* $\mathrm{fn}(P) \subseteq A$ *then* $A \vdash P \xrightarrow{\tau} Q$ *iff* $P \rightarrow Q$.

This give confidence that the labelled semantics carries enough information. The proof is somewhat delicate; it is sketched in [SV99b] and given in detail in [SV99a].

3 A Filtering Example

To demonstrate the use of box-π we give the definition of a wrapper that restricts the interface for user programs. In most operating systems, programs installed and run by a user enjoy the same access rights as the user, so if the user is allowed to open a socket and send data out on the network then so can any component. We idealize this scenario with the configuration below – an idealized single-user OS in which user Alice is executing a program P. Here the box around P stands for the operating system enforced user protection domain.

$alice[\ P\ ] \mid$	
$!...\overline{in}^{alice} x... \mid$	OS write on Alice's in port
$!\, out^{alice} x... \mid$	OS read from Alice's out port
$!\, net^{alice} x...$	OS read from Alice's net port

The OS provides three channels in, out and net, to respectively allow the user's program to read from and write to the terminal and to send data out on a network connection. The program P is executing within a box and so interacts with the OS using the $\uparrow$ tag – for example $P = in^{\uparrow} x.\overline{out}^{\uparrow} \langle x\, x \rangle$ receives a value from the terminal and then sends a pair of copies of the value back to the terminal.

To execute some untrusted code fragment Q, Alice may run the code in parallel with her other applications, perhaps as $alice[P \mid Q]$. But, this grants too much privilege to Q. In particular, if $Q = !\, in^{\uparrow} x.\overline{net}^{\uparrow} x$ then any terminal input may be redirected to the net. A wrapper is a box-π context which can provide fine-grain control of the behaviour of Q. For example, the filtering wrapper $\mathcal{W}_1$ of [SV99a] prevents Q from accessing the network:

$$\mathcal{W}_1(\_) \stackrel{def}{=} (\boldsymbol{\nu}\, a)\,(\ a[\ \_\ ] \ \mid\ !\, in^{\uparrow} x.\overline{in}^{a} x \ \mid\ !\, out^{a} x.\overline{out}^{\uparrow} x\ )$$

The system becomes $alice[P \mid \mathcal{W}_1(Q)]$. The untrusted code is placed in a box with a fresh name a, so $a \notin \mathrm{fn}(Q)$. In parallel with the box are two forwarders for in and out messages. The first, $!\, in^{\uparrow} x.\overline{in}^{a} x$, is a replicated input receiving values from the OS and sending them to a; the second is dual. Only these two processes can interact with a due to the scope of the restriction, so even when put in parallel with other code the wrapper guarantees that Q will not be able to send on net.

We show a small reduction sequence where $P = 0$ and $Q = in^{\uparrow} x.\overline{net}^{\uparrow} x$. Here B is the forwarders $!\, in^{\uparrow} x.\overline{in}^{a} x \mid !\, out^{a} x.\overline{out}^{\uparrow} x$.

$$\begin{array}{rl}
& \overline{in}^{alice} y \mid alice[P \mid \mathcal{W}_1(Q)] \\
\equiv & \overline{in}^{alice} y \mid alice[(\boldsymbol{\nu}\, a)(\ a[Q] \mid B\ )] \\
\rightarrow & alice[\overline{in}^{\uparrow} y \mid (\boldsymbol{\nu}\, a)(a[Q] \mid B)] \\
\equiv & alice[(\boldsymbol{\nu}\, a)(\overline{in}^{\uparrow} y \mid a[Q] \mid B)] \\
\rightarrow & alice[(\boldsymbol{\nu}\, a)(\overline{in}^{a} y \mid a[Q] \mid B)] \\
\rightarrow & alice[(\boldsymbol{\nu}\, a)(a[\overline{in}^{\uparrow} y \mid Q] \mid B)] \\
\rightarrow & alice[(\boldsymbol{\nu}\, a)(a[\overline{net}^{\uparrow} y] \mid B)] \\
\rightarrow & alice[(\boldsymbol{\nu}\, a)(\overline{net}^{a} y \mid a[0] \mid B)]
\end{array}$$

At the final step the output from Q is prevented from leaving the $alice$ box directly as B does not contain a forwarder for net. It is prevented from interaction with any P (although here P was empty) by the restriction on a.

4 The Unidirectional-flow Wrapper

There is a tension between the strength of communication primitive supported by a wrapper and the strength of the security property it can guarantee. The examples of the introduction and [SV99a] provide only asynchronous unordered communication between components, which would be awkward to use in most real systems. At the other extreme, synchronous communication introduces causal flows in both directions (the causal flow property we state in Section 5 would not hold in a synchronous calculus, so a more delicate property would be required – perhaps stating that there are only data-less acks from one component to another). There are two intermediate points – one can provide asynchronous ordered communication, as we do below, or use some form of weak acknowledgments, as in the NRL pump [KML96]. The former still guarantees an absence of information flow (albeit at the cost of maintaining an unbounded buffer) while the latter limits bandwidth of covert

$$
\begin{array}{lll}
\mathcal{F}(\_1, \_2) & = (\boldsymbol{\nu}\, a, b)\big(& a[\_1] \mid b[\_2] \mid \\
& & (\boldsymbol{\nu}\, \mathit{buff}, \mathit{full})\big(\\
& & \quad (\boldsymbol{\nu}\, \mathit{front}, \mathit{back})\big(\\
\text{(create FIFO buffer)} & & \quad\quad \overline{\mathit{buff}}^{\star}\langle \mathit{front}\ \mathit{back}\rangle \mid \\
\text{(connect } \mathit{from}^a \text{ to buffer)} & & \quad\quad !\, \mathit{from}^{a}(v\, r).(\boldsymbol{\nu}\, r')(\overline{\mathit{front}}^{\star}\langle v\, r'\rangle \mid r'^{\star}.\overline{r}^{a}) \mid \\
\text{(connect buffer to } \mathit{to}^b\text{)} & & \quad\quad !\, \mathit{back}^{\star}(v\, r).(\boldsymbol{\nu}\, r')(\overline{\mathit{to}}^{b}\langle v\, r'\rangle \mid r'^{b}.\overline{r}^{\star})\big) \mid \\
\text{(buffer code)} & & \quad !\, \mathit{buff}^{\star}(\mathit{front}\, \mathit{back}).\mathit{front}^{\star}(v\, r).(\overline{r}^{\star} \mid (\boldsymbol{\nu}\, \mathit{back}')(\overline{\mathit{buff}}^{\star}\langle \mathit{front}\, \mathit{back}'\rangle \\
& & \quad\quad\quad\quad \mid \overline{\mathit{full}}^{\star}\langle \mathit{back}'\ \mathit{back}\ v\rangle)) \mid \\
& & \quad !\, \mathit{full}^{\star}(\mathit{back}'\ \mathit{back}\ v).(\boldsymbol{\nu}\, r)(\overline{\mathit{back}}^{\star}\langle v\, r\rangle \mid r^{\star}.\mathit{back}'^{\star}(v'\ r').(\overline{r'}^{\star} \\
& & \quad\quad\quad\quad \mid \overline{\mathit{full}}^{\star}\langle \mathit{back}', \mathit{back}\ v'\rangle))\big) \mid \\
\text{(I/O forwarders)} & & !\, \mathit{in}_1{}^{\uparrow}x.\overline{\mathit{in}_1}^{a}x \mid !\, \mathit{out}_1{}^{a}x.\overline{\mathit{out}_1}^{\uparrow}x \mid \\
& & !\, \mathit{in}_2{}^{\uparrow}x.\overline{\mathit{in}_2}^{b}x \mid !\, \mathit{out}_2{}^{b}x.\overline{\mathit{out}_2}^{b}x\big)
\end{array}
$$

Figure 1. FIFO Pipeline Wrapper $\mathcal{F}$.

channels. In both cases, it is essential to be able to guarantee that the implementation of the communication primitives does actually have the desired flow property, this is what we set to do here.

In Figure 1 we give a wrapper $\mathcal{F}$ that takes two components and allows the first to communicate with the second by a first-in, first-out buffer. The wrapper has been written with care to avoid any information leak from the second component to the first. For simplicity both components have simple unordered input and output ports in_i and out_i to the environment; it would be routine to make these FIFO also. The wrapper is illustrated in Figure 2.
The interface to the wrapper is as follows. To write to the buffer a producer sends a value together with an acknowledgment channel to the wrapper (using a standard asynchronous π-calculus idiom). The wrapper inserts the value in a queue and acknowledges reception. For value v the producer may contain

$$(\boldsymbol{\nu}\, ack)(\overline{\mathit{from}}^{\uparrow}\langle v\, ack\rangle \ \mid\ ack^{\uparrow}...),$$

sending the value and a new acknowledgement channel ack to the wrapper and, in parallel, waiting for a reply before proceeding with its computation. On the receiver side, we may have a process that waits for a pair of a value and an ack channel:

$$to^{\uparrow}(z\, r).(\overline{r}^{\uparrow} \ \mid\ ...)$$

The name of the receiving channel is to; channel r is used to send the acknowledgement back to the wrapper. Thus a configuration where B stands for the body of the wrapper could be:

$$\begin{array}{l}(\boldsymbol{\nu}\, a, b)\big(\ a[\ (\boldsymbol{\nu}\, ack)(\overline{\mathit{from}}^{\uparrow}\langle v\, ack\rangle \ \mid\ ack^{\uparrow}.0)\ ] \ \mid \\ \quad b[\ to^{\uparrow}(z\, r).\overline{r}^{\uparrow}\ ] \ \mid B\ \big)\end{array}$$

The implementation of the wrapper is somewhat tricky, as we have to be careful not to introduce covert channels between the components. Within the wrapper there is a replicated input on *buff* that creates a new empty FIFO buffer and a replicated input on *full* that creates a new buffer cell containing a value. The key is to ensure that the acknowledgment to the first component not be dependent on any action performed by the second component. The glue process that connects the from^a channel to the buffer has a subprocess, $r'^{\star}.\overline{r}^{a}$, to send the ack to a. This small process itself expects an ack from the head of the buffer saying that the message was inserted in the queue. The buffer code $\mathit{front}^{\star}(v\, r).(\overline{r}^{\star}\ldots$ acks on r immediately, in parallel with placing the new message in a full buffer cell at the head of the queue. The asynchrony here is essential.

So far we have been vague about the statement of the properties that we expect wrappers to enforce. For $\mathcal{W}_1$ it may be clear from examination of the code and the semantics that the wrapper is satisfactory, but it is unclear exactly what properties are guaranteed. For $\mathcal{F}$ the situation is worse – even this simple wrapper is complex enough that a rigorous statement and proof of its security properties is essential; the user should not be required to examine the code of a wrapper in order to understand the security that it provides. We now turn to the task of formalizing these properties and developing the tools needed to prove them.

5 Colouring and Causal Flow

The intuitive property of $\mathcal{F}$ that we wish to express is that the second wrapped component should not be able to affect the first. In [SV99a] we expressed the intuitive property that one wrapped component does not causally affect another using a simple *coloured reduction semantics* for box-π. Output processes were annotated with sets of colours that record their causal histories – essentially the sets of principals that have affected them in the past – and the reduction semantics propagated this causal history data. In this paper

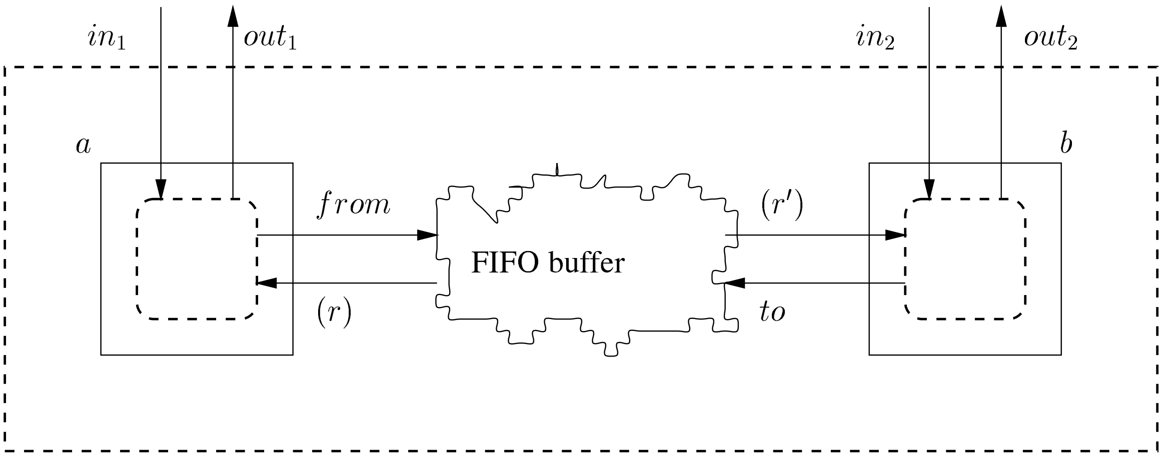


Figure 2. The FIFO Pipeline Wrapper Illustrated

we introduce also a coloured labelled transition semantics, allowing more direct statements of security properties of wrappers that interact with their environment. The coloured calculus is a trade-off – it captures less detailed causality information than the non-interleaving models studied in concurrency theory [WN95, BS95, DP95] but is much simpler; it captures enough information to express interesting security properties.

In [SV99a] we also expressed a number of other desirable properties of wrappers – that they *honestly* forward messages between component and environment, and that they *mediate* all communication between components. The latter, related to intransitive noninterference [RG99], was expressed using the coloured semantics. Two further information flow properties were expressed using the uncoloured LTS: *new name directionality* and *permutation*. They illustrate the wide range of precise properties which the intuitive statement might be thought to mean.

5.1 Colouring the Box-π Calculus

We take a set col of *colours* or *principals* (we use the terms interchangeably) disjoint from $\mathcal{N}$. Let $\mathsf{k}, \mathsf{p}, \mathsf{q}$ range over elements of col and $\mathsf{C}, \mathsf{D}, \mathsf{K}$ range over subsets of col. We define a coloured box-π calculus by annotating all outputs with sets of colours:

$$P \quad ::= \quad \mathsf{C}\!:\!\overline{x}^o v \;\Big|\; x^\iota p.P \;\Big|\; !\, x^\iota p.P \;\Big|\; n[P] \;\Big|\; 0 \;\Big|\; P \mid P' \;\Big|\; (\boldsymbol{\nu}\, x)P$$

If P is a coloured term we write $|P|$ for the term of the original syntax obtained by erasing all annotations. Conversely, for a term P of the original syntax $\mathsf{C} \circ P$ denotes the term with every particle coloured by C. For a coloured P we write $\mathsf{C} \bullet P$ for the coloured term which is as P but with C unioned to every set of colours occurring in it. We sometimes confuse p and the set $\{\mathsf{p}\}$. Let $\mathrm{pn}(P)$ be the set of colours that occur in P. We write CD for the union $\mathsf{C} \cup \mathsf{D}$.

In the coloured output $\mathsf{C}\!:\!\overline{x}^o v$ think of C as recording the causal history of the output particle – C is the set (possibly empty) of principals $\mathsf{p} \in \mathsf{C}$ that have affected the particle in the past. In an initial state all outputs might typically be coloured by singleton sets giving their actual principals, for example colouring the code of wrapper $\mathcal{F}$ and two wrapped components with different colours $\mathsf{w}, \mathsf{p}, \mathsf{q}$:

$$(\mathsf{w} \circ \mathcal{F})\,(\mathsf{p} \circ P \mid \mathsf{q} \circ Q)$$

The coloured reduction semantics is obtained by replacing the first four axioms of the uncoloured semantics by the rules

$$\begin{array}{ll}
n[\mathsf{C}\!:\!\overline{x}^{\uparrow} v \mid Q] \longrightarrow \mathsf{C}\!:\!\overline{x}^{\overline{n}} v \mid n[Q] & \text{(C Red Up)} \\
\mathsf{C}\!:\!\overline{x}^{n} v \mid n[Q] \longrightarrow n[\mathsf{C}\!:\!\overline{x}^{\overline{\uparrow}} v \mid Q] & \text{(C Red Down)} \\
\mathsf{C}\!:\!\overline{x}^{\overline{\iota}} v \mid x^\iota p.P \longrightarrow \mathsf{C} \bullet (\{^v/_p\}P) & \text{(C Red Comm)} \\
\mathsf{C}\!:\!\overline{x}^{\overline{\iota}} v \mid !\, x^\iota p.P \longrightarrow !\, x^\iota p.P \mid \mathsf{C} \bullet (\{^v/_p\}P) & \text{(C Red Repl)}
\end{array}$$

that propagate colour sets. The coloured calculus has essentially the same reduction behaviour as the original calculus:

Proposition 2 *For any coloured P we have $|P| \to Q$ iff $\exists P' \,.\, P \longrightarrow P' \wedge |P'| = Q$.*

The coloured labelled transitions have labels ℓ exactly as before. The coloured labelled transition relation has the form

$$A \vdash P \xrightarrow{\ell}_{\mathsf{C}} Q$$

where A is a finite set of names and $\mathrm{fn}(P) \subseteq A$; it should be read as 'in a state where the names A may be known to

$$\frac{}{A \vdash \mathsf{C}\!:\!\overline{x}^{o} v \xrightarrow{\overline{x}^{o} v}_{\mathsf{C}} 0}\ \text{(Out)}$$

$$\frac{}{A \vdash x^{\iota} p.P \xrightarrow{x^{\iota} v}_{\mathsf{C}} \mathsf{C}\bullet\{^{v}/_{p}\}P}\ (c)\ \text{(In)}$$

$$\frac{A \vdash P \xrightarrow{\ell}_{\mathsf{C}} P'}{A \vdash P \mid Q \xrightarrow{\ell}_{\mathsf{C}} P' \mid Q}\ \text{(Par)}$$

$$\frac{}{A \vdash \mathbf{!}\, x^{\iota} p.P \xrightarrow{x^{\iota} v}_{\mathsf{C}} \mathbf{!}\, x^{\iota} p.P \mid \mathsf{C}\bullet\{^{v}/_{p}\}P}\ (c)\ \text{(Repl)}$$

$$\frac{A \vdash P \xrightarrow{\overline{x}^{\overline{\gamma}} v}_{\mathsf{C}} P' \quad A \vdash Q \xrightarrow{x^{\gamma} v}_{\mathsf{C}} Q'}{A \vdash P \mid Q \xrightarrow{\tau}_{\emptyset} (\boldsymbol{\nu}\, \mathrm{fn}(x,v) - A)(P' \mid Q')}\ \text{(Comm)}$$

$$\frac{A \vdash P \xrightarrow{\overline{x}^{\uparrow} v}_{\mathsf{C}} P'}{A \vdash n[P] \xrightarrow{\tau}_{\emptyset} (\boldsymbol{\nu}\, \mathrm{fn}(x,v) - A)(\mathsf{C}\!:\!\overline{x}^{\overline{n}} v \mid n[P'])}\ \text{(Box-1)}$$

$$\frac{}{A \vdash n[P] \xrightarrow{x^{\overline{n}} v}_{\mathsf{C}} n[\mathsf{C}\!:\!\overline{x}^{\overline{\uparrow}} v \mid P]}\ \text{(Box-2)}$$

$$\frac{A \vdash P \xrightarrow{\tau}_{\mathsf{C}} P'}{A \vdash n[P] \xrightarrow{\tau}_{\mathsf{C}} n[P']}\ \text{(Box-3)}$$

$$\frac{A, x \vdash P \xrightarrow{\ell}_{\mathsf{C}} P'}{A \vdash (\boldsymbol{\nu}\, x)P \xrightarrow{\ell}_{\mathsf{C}} (\boldsymbol{\nu}\, x)P'}\ (a)\ \text{(Res-1)}$$

$$\frac{A, x \vdash P \xrightarrow{\overline{y}^{o} v}_{\mathsf{C}} P'}{A \vdash (\boldsymbol{\nu}\, x)P \xrightarrow{\overline{y}^{o} v}_{\mathsf{C}} P'}\ (b)\ \text{(Res-2)}$$

$$\frac{A \vdash P \xrightarrow{\ell}_{\mathsf{C}} P' \quad P' \equiv P''}{A \vdash P \xrightarrow{\ell}_{\mathsf{C}} P''}\ \text{(Struct)}$$

(a) The (Res-1) rule is subject to $x \notin \mathrm{fn}(\ell)$. (b) The (Res-2) rule is subject to $x \in \mathrm{fn}(v) - \mathrm{fn}(y, o)$, if o is $\star$, $\overline{\uparrow}$ or $\overline{n}$, and to $x \in \mathrm{fn}(y, v) - \mathrm{fn}(o)$ otherwise. (c) In the (In) and (Repl) axioms there is a side condition that $\{^{v}/_{p}\}P$ is well-defined. In all rules with conclusion of the form $A \vdash P \xrightarrow{\ell}_{\mathsf{C}} Q$ there is an implicit side condition $\mathrm{fn}(P) \subseteq A$. Symmetric versions of (Par) and (Comm) are elided.

Figure 3. Coloured Box-π Labelled Transition Semantics

P and its environment, process P can do ℓ, *coloured* C, to become Q'. Again C records causal history, giving all the principals which have directly or indirectly contributed to this action. The relation is defined as the smallest relation satisfying the rules in Figure 3. It coincides with the previous LTS and with the coloured reduction semantics in the following senses.

Proposition 3 *For any coloured P we have $A \vdash |P| \xrightarrow{\ell} Q$ iff $\exists \mathsf{C}, P'$. $A \vdash P \xrightarrow{\ell}_{\mathsf{C}} P' \wedge |P'| = Q$.*

Proposition 4 *For coloured P and Q, if $\mathrm{fn}(P) \subseteq A$ then $A \vdash P \xrightarrow{\tau}_{\emptyset} Q$ iff $P \to Q$.*

5.2 The Causal Flow Property

The property can now be stated. Say an *instantiation* of some binary wrapper $\mathcal{W}$ is an uncoloured process $\mathcal{W}(P,Q)$ where P and Q are uncoloured processes not containing the new-bound names scoping the holes of $\mathcal{W}$. Say $\mathcal{W}$ is a *pure binary wrapper* if for any instantiation and any transition sequence

$$A \vdash \mathcal{W}(P,Q) \xrightarrow{\ell_1} \ldots \xrightarrow{\ell_k} R$$

the labels ℓ_j have the form τ, $in_i{}^{\uparrow}v$, or $\overline{out_i}^{\uparrow}v$, for $i \in \{1,2\}$. It is easy to see that $\mathcal{F}$ is pure. Purity simply means that the wrapper has a fixed interface and thus simplifies the statement of the causal flow property.

Definition 1 (Causal flow property) *A pure binary wrapper $\mathcal{W}$ has the* causal flow property *if for any instantiation $\mathcal{W}(P,Q)$ and any coloured trace*

$$A \vdash \emptyset \circ \mathcal{W}(P,Q) \xrightarrow{\ell_1}_{\mathsf{C}_1} \ldots \xrightarrow{\ell_k}_{\mathsf{C}_k},$$

such that all input transitions $in_1{}^{\uparrow}v$ and $in_2{}^{\uparrow}v$ in $\ell_1..\ell_k$ are coloured with principal sets $\{\mathsf{p}\}$ and $\{\mathsf{q}\}$ respectively, we have $\ell_j = \overline{out_1}^{\uparrow}v$ implies that $\mathsf{q} \notin \mathsf{C}_j$.

This property forbids any causal flow from an input on in_2 to an output on out_1.

Different variants of the flow property, with different characteristics, can be stated – for example, to also prevent information in the initial state of Q affecting outputs on out_1 we could consider coloured traces

$$A \vdash (\emptyset \circ \mathcal{W})(\mathsf{p} \circ P, \mathsf{q} \circ Q) \xrightarrow{\ell_1}_{\mathsf{C}_1} \ldots \xrightarrow{\ell_k}_{\mathsf{C}_k}$$

This second definition still allows the Q to communicate with P but only on the condition that P does not perform any further output dependent on the communicated values. Forbidding Q affecting P at all (even if there are no inputs or outputs of either component) can be done with a slightly more intricate coloured semantics. There is no clear cut 'best' solution, yet the use of causal semantics allows succinct statement of the alternatives and eases the comparison of these different properties.

6 Causality Types

Verifying a causal flow property directly can be laborious, requiring a characterisation of the state space of a wrapper containing arbitrary components. We therefore introduce a type system that statically captures causal flows; a wrapper can be shown to satisfy the causal flow property simply by checking that it is well-typed. This section introduces the type system, gives its soundness theorems, and applies it to $\mathcal{F}$.

A simple type system for Box-π would have types

$$T ::= \mathbf{chan}\, T \;\big|\; \mathbf{box} \;\big|\; \langle T \,..\, T\rangle$$

for the types of channel names carrying T, box names, and tuples. We annotate the first two by sets K of principals and add a type $\mathbf{name}$, of arbitrary names, and $\top$, of arbitrary values, giving the *value types*

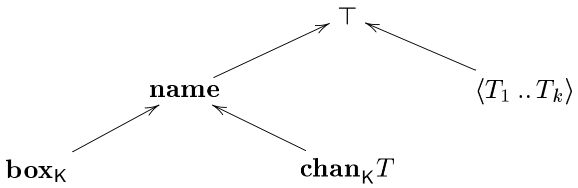
$$T ::= \mathbf{chan}_{\mathsf{K}} T \;\big|\; \mathbf{box}_{\mathsf{K}} \;\big|\; \langle T \,..\, T\rangle \;\big|\; \mathbf{name} \;\big|\; \top$$

If $x : \mathbf{chan}_{\mathsf{K}} T$ then x is the name of a channel carrying T; moreover, in an output process $\mathsf{C}\!:\!\overline{x}^{\star}v$ on x the typing rules will require $\mathsf{C} \subseteq \mathsf{K}$ – intuitively, such an output may have been causally affected only by the principals $\mathsf{k} \in \mathsf{K}$. In an input $x^{\iota}p.P$ on x the continuation P must therefore be allowed to be affected by any $\mathsf{k} \in \mathsf{K}$, so any output within P must be on a channel of type $\mathbf{chan}_{\mathsf{K}'} T$ with $\mathsf{K} \subseteq \mathsf{K}'$.

We are concerned with the encapsulation of possibly badly-typed components, so must allow a box $a[P]$ in a well-typed term to contain an untyped process P. The type system cannot be sensitive to the causal flows within such a box; it can only enforce an upper bound on the set of principals that can affect any part of the contents. If $a : \mathbf{box}_{\mathsf{K}}$ then a is a box name; the contents may have been causally affected only by $\mathsf{k} \in \mathsf{K}$.

We take *type environments* Γ to be finite partial functions from names to value types. The type system has two main judgments, $\Gamma \vdash v : T$ for values and $\Gamma \vdash P : \mathbf{proc}_{\mathsf{K}}$ for processes. The typing for processes records just enough information to determine when prefixing a process with an input is legitimate – if $P : \mathbf{proc}_{\mathsf{K}}$ then P can be prefixed by an input on a channel $x : \mathbf{chan}_{\mathsf{K}'} \langle\rangle$, to give $x^{\star}.P$, iff $\mathsf{K}' \subseteq \mathsf{K}$. Note, however, that a $P : \mathbf{proc}_{\mathsf{K}}$ may have been affected by (and so syntactically contain) $\mathsf{k} \notin \mathsf{K}$.

To type interactions between well-typed wrapper code and a badly-typed boxed component some simple subtyping is useful. We take the subtype order $T \leq T'$ as below, and write $\bigwedge\{\, T_i \mid i \in 1..k \,\}$ for the greatest lower bound of $T_1, .., T_k$, where this exists.



The complete type system is given in Figure 4; we now explain the key aspects by giving some admissible typing rules.

Basic Flow Typing Consider the type environment $x : \mathbf{chan}_{\mathsf{K}}\langle\rangle$, $y : \mathbf{chan}_{\mathsf{L}}\langle\rangle$ and the reduction

$$\mathsf{C}:\overline{x}^{\star} \mid x^{\star}.\mathsf{D}:\overline{y}^{\star} \to (\mathsf{C}\cup\mathsf{D}):\overline{y}^{\star}$$

During the reduction the output $\overline{y}^{\star}$ on y is causally affected by the output on x – the right-hand process term $(\mathsf{C}\cup\mathsf{D}):\overline{y}^{\star}$ records that the output on y has been (indirectly) affected by all the principals that had affected the output on x. For the left process to be well-typed we must clearly require $\mathsf{C} \subseteq \mathsf{K}$ and $\mathsf{D} \subseteq \mathsf{L}$; for the right process to be well-typed we need also $\mathsf{C} \subseteq \mathsf{K}$, to guarantee this the typing rules require $\mathsf{K} \subseteq \mathsf{L}$. The relevant admissible rules are below.

$$\frac{\begin{array}{l}\Gamma \vdash x : \mathbf{chan}_{\mathsf{K}} T\\ \Gamma \vdash v : T\\ \mathsf{C} \subseteq \mathsf{K}\end{array}}{\Gamma \vdash \mathsf{C}:\overline{x}^{\star}v : \mathbf{proc}_{\mathsf{K}}} \qquad \frac{\begin{array}{l}\Gamma \vdash x : \mathbf{chan}_{\mathsf{K}} T\\ \Gamma, y : T \vdash P : \mathbf{proc}_{\mathsf{K}''}\\ \mathsf{K} \subseteq \mathsf{K}''\end{array}}{\Gamma \vdash x^{\star}y.P : \mathbf{proc}_{\mathsf{K}}}$$

Now consider also $y : \mathbf{chan}_{\mathsf{L}'}\langle\rangle$ and the process

$$\mathsf{C}:\overline{x}^{\star} \mid x^{\star}.\big(\mathsf{D}:\overline{y}^{\star} \mid \mathsf{D}':\overline{y'}^{\star}\big)$$

Here both the output on y and that on y' must be affectable by C, so the typing rule for parallel must take the intersection of allowed-cause sets:

$$\frac{\Gamma \vdash P : \mathbf{proc}_{\mathsf{K}} \quad \Gamma \vdash Q : \mathbf{proc}_{\mathsf{K}'}}{\Gamma \vdash P \mid Q : \mathbf{proc}_{\mathsf{K}\cap\mathsf{K}'}}$$

The examples above involve only communication within a wrapper, with tag $\star$. Communication between a wrapper and its parent, with tag $\uparrow$, has the same typing rules, as the parent is presumed well-typed.

Channel Passing Channel passing involves no additional complication. Consider the type environment $\Gamma = z : \mathbf{chan}_{\mathsf{K}''}\langle\rangle, x : \mathbf{chan}_{\mathsf{K}}\mathbf{chan}_{\mathsf{K}''}\langle\rangle$, and the reduction

$$\mathsf{C}:\overline{x}^{\star}z \mid x^{\star}y.\mathsf{D}:\overline{y}^{\star} \to (\mathsf{C}\cup\mathsf{D}):\overline{z}^{\star}$$

The left-hand process is typable using the rules above if $\mathsf{C} \subseteq \mathsf{K}$ for the x output, $\mathsf{D} \subseteq \mathsf{K}''$ for the y output, and $\mathsf{K} \subseteq \mathsf{K}''$ for the input, using $\Gamma, y : \mathbf{chan}_{\mathsf{K}''}\langle\rangle \vdash \mathsf{D} : \overline{y}^{\star} : \mathbf{proc}_{\mathsf{K}''}$. Together these imply $(\mathsf{C}\cup\mathsf{D}) \subseteq \mathsf{K}''$, so the right-hand process is well-typed.

Interacting with a box (at $\top$) As discussed above, the contents of a box may be badly-typed, yet a wrapper must still be able to interact with them. The simplest case is that in which a wrapper sends and receives values that it considers to be of type $\top$; we consider more general communication in the next paragraph. The typing rule for boxes requires only that the principals $\mathrm{pn}(P)$ syntactically occurring within the contents P of a box are contained in the permitted set and that P's free names are all declared in the type environment.

$$\frac{\begin{array}{l}\Gamma \vdash a : \mathbf{box}_{\mathsf{K}}\\ \mathrm{pn}(P) \subseteq \mathsf{K}\\ \mathrm{fn}(P) \subseteq \mathrm{dom}(\Gamma)\end{array}}{\Gamma \vdash a[P] : \mathbf{proc}_{\mathsf{K}}}$$

Consider sending to and receiving from a box $a : \mathbf{box}_{\mathsf{K}}$.

$$\mathsf{C}:\overline{x}^{a}v \mid a[P] \mid z^{a}y.Q$$

For the output to be well-typed we must insist only that $\mathsf{C} \subseteq \mathsf{K}$; for the input to be well-typed Q must be allowed to be affected by any principal that might have affected the contents P.

$$\frac{\begin{array}{l}\Gamma \vdash a : \mathbf{box}_{\mathsf{K}}\\ \Gamma \vdash x : \mathbf{name}\\ \Gamma \vdash v : \top\\ \mathsf{C} \subseteq \mathsf{K}\end{array}}{\Gamma \vdash \mathsf{C}:\overline{x}^{a}v : \mathbf{proc}_{\mathsf{K}}} \qquad \frac{\begin{array}{l}\Gamma \vdash a : \mathbf{box}_{\mathsf{K}}\\ \Gamma \vdash x : \mathbf{chan}_{\mathsf{K}'}\top\\ \Gamma, y : \top \vdash P : \mathbf{proc}_{\mathsf{K}''}\\ \mathsf{K} \subseteq \mathsf{K}' \subseteq \mathsf{K}''\end{array}}{\Gamma \vdash x^{a}p.P : \mathbf{proc}_{\mathsf{K}'}}$$

Interacting with a box (at any transmissible S) More generally, a wrapper may receive from a box tuples containing names which are to be used *for communicating with the box* as channel names, for example

$$x^{a}(v\,r).\big(\mathsf{C}:\overline{r}^{a} \mid \ldots\big)$$

receives a value v and name r from box a and uses r to send an ack back into a. This necessarily involves some run-time typechecking, as the box may send a tuple instead of a name for r. There is a design choice here: how strong should this typechecking be? Requiring an implementation to maintain a run-time record of the types of all names would be costly, so we check only the structure of values received from boxes. We suppose the run-time representations of values allow names (bit-patterns of some fixed length) and tuples to be distinguished, and the number of items in a tuple to be determined, but no more (so e.g. $x : \mathbf{chan}_{\mathsf{K}}T$ and $y : \mathbf{box}_{\mathsf{L}}$ will both be represented as bit patterns of the same

Patterns:

$$\frac{}{\vdash \_ : T \rhd \emptyset} \qquad \frac{}{\vdash x : T \rhd x : T} \qquad \frac{\vdash p_1 : T_1 \rhd \Delta_1 \;..\; \vdash p_k : T_k \rhd \Delta_k}{\vdash (p_1 \,..\, p_k) : \langle T_1 \,..\, T_k \rangle \rhd \Delta_1, .., \Delta_k}$$

Values:

$$\frac{}{\Gamma, x : T \vdash x : T} \qquad \frac{\Gamma \vdash v_1 : T_1 \;..\; \Gamma \vdash v_k : T_k}{\Gamma \vdash \langle v_1 \,..\, v_k \rangle : \langle T_1 \,..\, T_k \rangle} \qquad \frac{\mathrm{fn}(v) \subseteq \mathrm{dom}(\Gamma)}{\Gamma \vdash v : \top} \qquad \frac{T \text{ atomic}}{\Gamma, x : T \vdash x : \mathbf{name}}$$

Processes:

$$\frac{\begin{array}{l} o \in \{\star, \uparrow, \overline{\uparrow}\} \\ \Gamma \vdash x : \mathbf{chan}_{\mathsf{K}} T \\ \Gamma \vdash v : T \\ \mathsf{C} \subseteq \mathsf{K} \end{array}}{\Gamma \vdash \mathsf{C} : \overline{x}^{o} v \,:\, \mathbf{proc}_{\mathsf{K}}} \ (\text{Out-}\star, \uparrow, \overline{\uparrow}) \qquad \frac{\begin{array}{l} \iota \in \{\star, \uparrow\} \\ \Gamma \vdash x : \mathbf{chan}_{\mathsf{K}} T \\ \vdash p : T \rhd \Delta \\ \Gamma, \Delta \vdash P : \mathbf{proc}_{\mathsf{K}} \end{array}}{\Gamma \vdash x^{\iota} p.P : \mathbf{proc}_{\mathsf{K}}} \ (\text{In-}\star, \uparrow)$$

$$\frac{\begin{array}{l} o \in \{a, \overline{a}\} \\ \Gamma \vdash a : \mathbf{box}_{\mathsf{K}} \\ \Gamma \vdash x : \mathbf{name} \\ \Gamma \vdash v : \top \\ \mathsf{C} \subseteq \mathsf{K} \end{array}}{\Gamma \vdash \mathsf{C} : \overline{x}^{o} v \,:\, \mathbf{proc}_{\mathsf{K}}} \ (\text{Out-}a, \overline{a}) \qquad \frac{\begin{array}{l} \Gamma \vdash a : \mathbf{box}_{\mathsf{K}'} \\ \Gamma \vdash x : \mathbf{chan}_{\mathsf{K}} S \\ \vdash p : S \rhd \Delta \\ \Gamma, \Delta \vdash P : \mathbf{proc}_{\mathsf{K}} \\ \mathsf{K}' \subseteq \mathsf{K} \\ \Delta \text{ flat} \\ P \text{ tests all names of type } \mathbf{name} \text{ in } \Delta \\ p \text{ contains no wildcards} \end{array}}{\Gamma \vdash x^{a} p.P : \mathbf{proc}_{\mathsf{K}}} \ (\text{In-}a)$$

$$\frac{\begin{array}{l} \Gamma \vdash P : \mathbf{proc}_{\mathsf{K}} \\ \Gamma \vdash Q : \mathbf{proc}_{\mathsf{K}'} \end{array}}{\Gamma \vdash P \mid Q : \mathbf{proc}_{\mathsf{K} \cap \mathsf{K}'}} \ (\text{Par}) \qquad \frac{\begin{array}{l} \Gamma \vdash n : \mathbf{box}_{\mathsf{K}} \\ \mathrm{pn}(P) \subseteq \mathsf{K} \\ \mathrm{fn}(P) \subseteq \mathrm{dom}(\Gamma) \end{array}}{\Gamma \vdash n[P] : \mathbf{proc}_{\mathsf{K}}} \ (\text{Box})$$

$$\frac{}{\Gamma \vdash 0 : \mathbf{proc}_{\mathsf{K}}} \ (\text{Nil}) \qquad \frac{\begin{array}{l} \Gamma, x : T \vdash P : \mathbf{proc}_{\mathsf{K}} \\ T \text{ atomic} \end{array}}{\Gamma \vdash (\boldsymbol{\nu}\, x) P : \mathbf{proc}_{\mathsf{K}}} \ (\text{Res})$$

$$\frac{\begin{array}{l} \Gamma \vdash P : \mathbf{proc}_{\mathsf{K}'} \\ \mathsf{K} \subseteq \mathsf{K}' \end{array}}{\Gamma \vdash P : \mathbf{proc}_{\mathsf{K}}} \ (\text{Spec})$$

The replicated input rules are similar to the input rules. The predicate 'P tests all names of type $\mathbf{name}$ in Δ' is defined to be true iff for all $y : \mathbf{name}$ in Δ, y occurs free in channel or box position within P.

Figure 4. Coloured Box-π Typing

length). We introduce the supertype $\mathbf{name}$ of $\mathbf{chan}_\mathsf{K} T$ and $\mathbf{box}_\mathsf{L}$, and allow a wrapper to receive only values of the *transmissible types*

$$S ::= \top \mid \mathbf{name} \mid \langle S \,..\, S \rangle$$

To send a value to a box by $\mathsf{C}:\overline{x}^a v$ it is necessary only for x to be of type $\mathbf{name}$.

The operational semantics expresses this run-time typechecking by means of the condition that $\{^v/_p\}P$ is well-defined in the reduction communication rule and the labelled-transition input rules – for example, $\{^{\langle z\, z\rangle}/_x\}\mathsf{C}:\overline{x}^\star$ is not well-defined, as the syntax does not allow a tuple to occur in channel-name position of an output. We would like to ensure that run-time typechecking is only required when receiving values from a box, i.e. that for communication within a wrapper or between a wrapper and its parent such a substitution is always well-defined. This is guaranteed by requiring a box input prefix to immediately test all parts of a received value that are assumed of type $\mathbf{name}$ – in typing an input $x^a p.P$ the type environment Δ derived from the pattern p must contain no tuples, and all $x:\mathbf{name}$ in Δ must be used within P as a channel or box. For example, if $a:\mathbf{box}_\mathsf{K}$ and $x:\mathbf{chan}_\mathsf{K}\langle\mathbf{name}\,\mathbf{name}\rangle$ then

$$x^a(y\; z).\big(\mathsf{K}:\overline{y}^a \mid \mathsf{K}:\overline{z}^a\big)$$

is well-typed as the pattern $(y\; z)$ completely decomposes values of type $\langle\mathbf{name}\,\mathbf{name}\rangle$ and both y and z are used as channels in $\mathsf{K}:\overline{y}^a \mid \mathsf{K}:\overline{z}^a$. On the other hand

$$x^a w.\overline{x}^\star w$$

is not, as it may receive (for example) a triple from the box, leading to a later run-time error within the wrapper. The type system is conservative in also excluding $x^a(y\; z).\big(\mathsf{K}:\overline{y}^a\big)$. Say a type is *atomic* if it is of the form $\mathbf{name}$, $\mathbf{chan}_\mathsf{K} T$ or $\mathbf{box}_\mathsf{K}$ and *flat* if it is of the form $\top$, $\mathbf{name}$, $\mathbf{chan}_\mathsf{K} T$, or $\mathbf{box}_\mathsf{K}$. Say Γ is atomic or flat if all types in $\mathrm{ran}(\Gamma)$ are. The atomic types are those which can be dynamically extended using restriction. We consider dynamics (reductions and labelled transitions) only for processes with respect to atomic typing contexts; the definitions ensure that an extruded name can always be taken to be of an atomic type. The calculus has no basic data types, e.g. a type of integers, that are not dynamically extensible. This makes the type system a little degenerate.

The rest The typing rules for nil and restriction are straightforward; there is also a specialisation rule allowing some permitted affectees of a process to be forgotten.

$$\frac{}{\Gamma \vdash 0:\mathbf{proc}_\mathsf{K}} \qquad \frac{\Gamma, x:T \vdash P:\mathbf{proc}_\mathsf{K} \quad T \text{ atomic}}{\Gamma \vdash (\boldsymbol{\nu}\, x)P:\mathbf{proc}_\mathsf{K}} \qquad \frac{\Gamma \vdash P:\mathbf{proc}_{\mathsf{K}'} \quad \mathsf{K} \subseteq \mathsf{K}'}{\Gamma \vdash P:\mathbf{proc}_\mathsf{K}}$$

6.1 Soundness

We wish to infer properties of the coloured input/output behaviour of wrappers from the soundness of the type system, and therefore need a subject reduction result which refers not only to reductions (equivalently, τ transitions) but also to input/output transitions. Define typed labelled transitions by

$$\Gamma \vdash_\mathsf{K} P \xrightarrow{\ell}_\mathsf{C} Q \quad \text{iff} \quad \big(\Gamma \text{ atomic} \wedge \Gamma \vdash P:\mathbf{proc}_\mathsf{K} \wedge \mathrm{dom}(\Gamma) \vdash P \xrightarrow{\ell}_\mathsf{C} Q\big)$$

The subject reduction theorem for ℓ an output $\overline{x}^o v$ should state that x, o, v and Q have suitable types; the theorem for ℓ an input should state that if ℓ can be typed then Q can. The result is complicated by the fact that box-π is a calculus with new name generation, so new names can be extruded and intruded. Type environments for these names are calculated as follows. For a type environment Γ, with Γ atomic, and a value v extruded at type T define the type environment $tc(\Gamma, v, T)$ for new names in v as follows.

$$\begin{array}{lll}
tc(\Gamma, x, T) & = x:T & \text{if } x \notin \mathrm{dom}(\Gamma) \\
 & & \text{and } T \text{ atomic} \\
tc(\Gamma, x, \top) & = x:\mathbf{name} & \text{if } x \notin \mathrm{dom}(\Gamma) \\
tc(\Gamma, x, T) & = \emptyset & \text{if } \Gamma \vdash x:T \\
tc(\Gamma, \langle v_1 \,..\, v_k\rangle, \top) & = \bigwedge_{1..n} tc(\Gamma, v_i, \top) & \\
tc(\Gamma, \langle v_1 \,..\, v_k\rangle, \langle T_1 \,..\, T_k\rangle) & = \bigwedge_{1..n} tc(\Gamma, v_i, T_i) & \\
tc(\Gamma, v, T) \text{ undefined elsewhere} & &
\end{array}$$

Here $\bigwedge_{i \in 1..k} \Gamma_i$ is the type environment that maps each x in some $\mathrm{dom}(\Gamma_i)$ to $\bigwedge\{T \mid \exists i \,.\, x:T \in \Gamma_i\}$, where all of these are defined. $\bigwedge_{i \in 1..k} \Gamma_i$ is undefined otherwise. Note that in the $\top$ case the $tc(\Gamma, v_i, \top)$ will necessarily all be well-defined and will be consistent. To see the need for $\bigwedge$, consider $\Gamma = c:\mathbf{chan}_\mathsf{K}\langle\mathbf{box}_\mathsf{K}\,\mathbf{name}\rangle$ and $P = (\boldsymbol{\nu}\, x)\overline{c}^\star\langle x\, x\rangle$. P has an extrusion transition with value $\langle x\, x\rangle$; the type context $tc(\Gamma, \langle x\, x\rangle, \langle\mathbf{box}_\mathsf{K}\,\mathbf{name}\rangle)$ should be well-defined and equal to $x:\mathbf{box}_\mathsf{K}$.

Further, the type system involves subtyping, so $tc(\Gamma, v, T)$ can only be used as a bound on the extruded/intruded type environments. Say $\Gamma \leq \Gamma'$ iff $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma')$ and $\forall x \in \mathrm{dom}(\Gamma) \,.\, \Gamma(x) \leq \Gamma'(x)$.

We can now state the subject reduction result. For output tags $\{\star, \overline{\uparrow}\}$ and $\uparrow$ the name x is guaranteed to have a channel type and v the type carried; for a and $\overline{a}$ they are only guaranteed to be a $\mathbf{name}$ and a value of type $\top$. $\{\star, \overline{\uparrow}\}$ and $\overline{a}$ are communication tags, so x cannot be extruded, whereas $\uparrow$ and a are movement tags, so x may be extruded. By convention we elide a conjunct that $tc(...)$ is defined wherever it is mentioned.

Theorem 5 (Subject Reduction) *If* $\Gamma \vdash_{\mathsf{K}} P \xrightarrow{\overline{x}^{o} v}_{\mathsf{C}} Q$ *then*

case $o \in \{\star, \overline{\uparrow}\}$**:** *for some* K', T *we have* $\mathsf{C} \subseteq \mathsf{K}'$, $\Gamma \vdash x : \mathbf{chan}_{\mathsf{K}'} T$, *and there exists* $\Theta \leq tc(\Gamma, v, T)$ *such that* $\Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

case $o = \uparrow$**:** *for some* K', T *we have* $\mathsf{C} \subseteq \mathsf{K}'$ *and there exists* $\Theta \leq tc(\Gamma, \langle x\, v\rangle, \langle \mathbf{chan}_{\mathsf{K}'} T\, T\rangle)$ *such that* $\Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

case $o = a$**:** *for some* K' *we have* $\mathsf{C} \subseteq \mathsf{K}'$, $\Gamma \vdash a : \mathbf{box}_{\mathsf{K}'}$, *and there exists a type environment* $\Theta \leq tc(\Gamma, \langle x\, v\rangle, \langle \mathbf{name}, \top\rangle)$ *such that* $\Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

case $o = \overline{a}$**:** *for some* K' *we have* $\mathsf{C} \subseteq \mathsf{K}'$, $\Gamma \vdash a : \mathbf{box}_{\mathsf{K}'}$, $\Gamma \vdash x : \mathbf{name}$, *and there exists* $\Theta \leq tc(\Gamma, v, \top)$ *such that* $\Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

If $\Gamma \vdash_{\mathsf{K}} P \xrightarrow{x^{\gamma} v}_{\mathsf{C}} Q$ *then*

case $\gamma \in \{\star, \uparrow\}$**:** *for some* K', T *we have* $\Gamma \vdash x : \mathbf{chan}_{\mathsf{K}'} T$. *If moreover* $\mathsf{C} \subseteq \mathsf{K}'$ *and* $\Theta \leq tc(\Gamma, v, T)$ *then* $\Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

case $\gamma = a$**:** *for some* $\mathsf{K}' \subseteq \mathsf{K}''$, *and* S *we have* $\Gamma \vdash a : \mathbf{box}_{\mathsf{K}'}$, $\Gamma \vdash x : \mathbf{chan}_{\mathsf{K}''} S$, $tc(\Gamma, v, S)$ *well-defined, and* $\mathrm{ran}(tc(\Gamma, v, S)) \subseteq \{\mathbf{name}\}$. *If moreover* $\mathsf{C} \subseteq \mathsf{K}''$ *and* $\Theta \leq tc(\Gamma, v, S)$ *then* $\Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

case $\gamma = \overline{a}$**:** *for some* K' *we have* $\Gamma \vdash a : \mathbf{box}_{\mathsf{K}'}$. *If moreover* $\mathsf{C} \subseteq \mathsf{K}'$ *and we have* $\Theta \leq tc(\Gamma, \langle x\, v\rangle, \langle \mathbf{name}\, \top\rangle)$ *then* $\Gamma, \Theta \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

If $\Gamma \vdash_{\mathsf{K}} P \xrightarrow{\tau}_{\mathsf{C}} Q$ *then* $\mathsf{C} = \emptyset$ *and* $\Gamma \vdash Q : \mathbf{proc}_{\mathsf{K}}$.

A run-time error for box-π is a process in which a potential communication fails because the associated substitution is not defined. More precisely, P contains a run-time error if it contains subterms $\overline{x}^{\gamma} v$ and $x^{\gamma} p.P$ in parallel (and not under an input prefix) and $\{^{v}/_{p}\}P$ is not defined. In a well-typed process run-time errors can only occur within boxes (whose contents are untyped) or at communications from a box to the wrapper. Internal transitions of the wrapper and communications between the wrapper and its parent therefore do not require dynamic typechecking.

Theorem 6 (Limited Runtime Errors)
If $\Gamma \vdash P : \mathbf{proc}_{\mathsf{K}}$, $P \equiv (\boldsymbol{\nu}\, x_1 \ldots x_n)(\overline{x}^{\gamma} v \mid x^{\gamma} p.P' \mid Q)$, Γ *atomic,* P' *does not contain a box and* $\gamma \in \{\star, \uparrow\}$ *then* $\{^{v}/_{p}\}P$ *is well-defined. Similarly for replicated input.*

6.2 Typing the Unidirectional-flow Wrapper

Finally, we can show that instantiations of $\mathcal{F}$ are well-typed and use the subject reduction theorem to conclude that $\mathcal{F}$ has the causal flow property.

Theorem 7 ($\mathcal{F}$ typing) *If*

$$\begin{array}{rcll} \Gamma & = & in_1 : \mathbf{chan}_{\{\mathsf{p}\}} \top, & out_1 : \mathbf{chan}_{\{\mathsf{p}\}} \top, \\ & & in_2 : \mathbf{chan}_{\{\mathsf{q}\}} \top, & out_2 : \mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}} \top, \\ & & \mathit{from} : \mathbf{chan}_{\{\mathsf{p}\}} \langle \top\, \mathbf{name} \rangle, & \\ & & \mathit{to} : \mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}} \langle \top\, \mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}} \langle\rangle \rangle, & \\ & & \Gamma_1 & \end{array}$$

and also $\mathrm{fn}(P, Q) \subseteq \mathrm{dom}(\Gamma) - \{a, b\}$
then $\Gamma \vdash \emptyset \circ \mathcal{F}(P, Q) : \mathbf{proc}_{\mathsf{p}}$.

The proof of this involves type assumptions for the new-bound names of $\mathcal{F}$ as follows.

$$\begin{array}{rl} a{:}\mathbf{box}_{\{\mathsf{p}\}} & \\ b{:}\mathbf{box}_{\{\mathsf{p},\mathsf{q}\}} & \\ \mathit{buff}{:}\mathbf{chan}_{\{\mathsf{p}\}} \langle & \mathbf{chan}_{\{\mathsf{p}\}} \langle \top\, \mathbf{chan}_{\{\mathsf{p}\}} \langle\rangle \rangle \\ & \mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}} \langle \top\, \mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}} \langle\rangle \rangle \rangle \\ \mathit{full}{:}\mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}} \langle & \mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}} \langle \top\, \mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}} \langle\rangle \rangle \\ & \mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}} \langle \top\, \mathbf{chan}_{\{\mathsf{p},\mathsf{q}\}} \langle\rangle \rangle \\ & \top \rangle \end{array}$$

A straightforward induction on trace lengths using the Subject Reduction theorem then proves the desired causal flow result:

Theorem 8 *Wrapper* $\mathcal{F}$ *has the causal flow property.*

7 Discussion

Policy enforcement mechanisms: Wrappers impose security policies on components for which it is impractical to analyze the internal structure, e.g. where only untyped object code is available.

Several alternative approaches are possible, differing in the level of trust required, the flexibility of the security policy enforced, and their costs to component producers and users. Code signing and Java-style sandboxing have low cost but cannot enforce flexible policies – signed components may behave in arbitrary ways whereas sandboxed components should not be able to interact with each other at all. Code signing requires the user to have total trust in the component producers – not just in their intent, but also in their ability to produce bug-free components. Sandboxing requires no trust, but the lack of any interaction is often too restrictive. More delicate policies can be enforced by shipping code together with data allowing the user to typecheck it in a security-sensitive type system [VSI96, HR98], or to check a proof of a security-relevant behavioural property [NL98]. In the long term these seem likely to be the best approaches, but they require component producers to invest effort and to conform to a common standard for types or proofs – in the short term this is prohibitive. Shifting the burden of proof to the user, by performing type inference or static analysis of downloaded code, seems impractical given only the object code, which may not have been written with security in mind and so not conform to any reasonable type system. In contrast, wrappers have been shown to have low-cost – none to the producer and only a small run-time cost to the user [FBF99]. They allow more flexible interaction than sandboxing, albeit coarser-grain policies than proof-carrying components or security-type-checked components.

Information flow properties: The causal flow property is related to the property, studied in many contexts, that there is no information flow from a high to a low security level (though most work addresses components, which may have the property, rather than wrappers, which may enforce it on subcomponents). The literature contains a range of definitions that aim to capture this intuition in some particular setting; the formalisations vary widely. A basic choice is whether the property is stated purely extensionally, in terms of a semantics that describes only the input/output behaviour of a system, or using a more intensional semantics. A line of work on Non-Interference, summarised in [McL94], takes an extensional approach, stating properties in terms of the traces of input and output events of a system. Related definitions, adapted to a programming language setting, are used in [VSI96, HR98]. In the presence of nondeterminism, however, non-interference becomes problematic – as discussed in [VS98], the property may only be meaningful given probabilistic scheduling, which has a high runtime cost.

We believe that the basic difficultly is that the intuitive property is an *intensional* one – the notion of one component affecting another depends on some understanding of how components interact; a precise statement requires a semantics that captures some aspects of internal execution, not just input/output behaviours. This might be denotational or operational. Intensional denotational semantics have been used in the proofs (and, in the last, statements) of non-interference properties in [HR98, ABHR99, SS99], which use a logical relations proof and PER-based models. [VS98] and [SS99] go on to consider probabilistic properties.

For wrappers, it is important that the end-user be able to understand the security that they provide as clearly as possible. We therefore wish to use as lightweight a semantics as possible, as this must be understood before any security property stated using it, and so adopt an annotated operational semantics (developing a satisfactory denotational semantics of box-π, dealing with name creation, boxes, and untyped components, would be a challenging research problem in its own right). In a sequential setting annotated operational semantics have been used by [ZGM99]; see also [LR98]. The definition of the coloured semantics for box-π seems unproblematic, but in general one might validate an annotated semantics by relating it to a lower-level execution model (as mentioned below).

Neglecting boxing and wrappers for the moment, considering simply π-processes, we believe that intensional properties stated in terms of causal flow will generally imply properties stated purely in terms of trace-sets. As a starting point, we show that our type system implies a non-interference property (similar to the permutation property of [SV99b], but for processes rather than wrappers) in a particular case. We prove that an output on a 'low' channel can always be permuted before an input on a 'higher' channel (with respect to the lattice of sets of colours).

Proposition 9 *If* $\mathsf{L} \subsetneq \mathsf{H}$ *and* $\{h : \mathbf{chan}_{\mathsf{H}} U,\ l : \mathbf{chan}_{\mathsf{L}} V\} \vdash P : \mathbf{proc}_{\emptyset}$ *then*

$$\{h, l\} \vdash P \xrightarrow{h^\star u} \xrightarrow{\bar{l}^\star v} Q \quad \textit{implies} \quad \{h, l\} \vdash P \xrightarrow{\bar{l}^\star v} \xrightarrow{h^\star u} Q.$$

Proof (Sketch) One can first show that $\emptyset \circ P$ has coloured transitions with the input coloured H and the output by some C. By subject reduction $\mathsf{C} \subseteq \mathsf{L}$. Analysing the form of P with Lemmas 21,20 from [SV99a], and using $\mathsf{L} \subsetneq \mathsf{H}$, shows that the output term in P is not prefixed by the input, so the transitions can be permuted. $\square$

Information flow type systems: The type system differs from previous work [VSI96, VS98, PØ97] primarily in handling badly typed components. Necessarily, it does not provide fine-grain tracking of information flow through these components. It also handles nondeterminism, new name creation and channel passing. Precise comparisons with related type systems are difficult as the languages involved differ widely. One can, however, embed fragments of these languages into box-π (noting that this only exploits the fully-typed part of our calculus). For example, in the work of Smith and Volpano [SV98] an assignment to a low security variable can follow an assignment to a high variable – the program `h:=3;l:=1` is well-typed. The natural translation of this program in box-π would be

$$\overline{h}^\star 0 \mid \overline{l}^\star 0 \quad \mid \quad h^\star y.(\overline{h}^\star 3 \mid l^\star y.\overline{l}^\star 1)$$

with an initial store assigning 0 to h and l. This would not be well-typed in the system of this paper, taking $h : \mathbf{chan}_{\{\mathsf{H},\mathsf{L}\}}\mathbf{Int}$, $l : \mathbf{chan}_{\{\mathsf{L}\}}\mathbf{Int}$ and a new base type $\mathbf{Int}$. Here the low assignment is causally dependent on the high, even though no high information can leak. On the other hand a box-π encoding of branches would not forbid high variable guards.

Causal flow is a robust and straightforward property; it can be enforced by a remarkably simple type system. But, as the example above shows, it is sometimes overconstraining. We envisage that in a large system the bulk of the code will be typeable in a secure type system, a small portion will be in clearly-identified unsafe modules that are subject only to conventional typechecking, and a small portion (any untrusted code) will be encapsulated in wrappers. Automatic type inference would be required to relieve the burden of adding security annotations to all declarations.

8 Conclusion

The issue of securely composing untrusted or partially trusted components has great practical relevance. In this paper we have studied techniques for formally proving that software wrappers – the glue between components – actually enforce user-specified information flow constraints. We have defined a coloured operational semantics for a concurrent wrapper language. By keeping track of all the principals that have affected a process in the semantics it becomes easy to formulate clear statements of information flow properties. To prove that particular wrappers are secure, we defined a causal type system and so only need show that the wrappers are well typed.

Throughout the paper we focussed on wrapper properties – the calculus, statement of security properties and type system are all designed specifically for wrappers – but we believe similar techniques are applicable to other situations in which interaction must be controlled but not completely excluded, for example in isolating a security-critical kernel of a single application, or in controlling interactions between packets in an active network. Allowing untyped code fragments in otherwise typed programs gives a way to loosen security restrictions when necessary.

In future work we intend to integrate the causal type system with a lower-level semantics for object code, such as the typed assembly language of [GM99]. We also intend to address the issue of type inference of security levels and the statements of properties involving dynamic changes in information flow policy.

Acknowledgements We would like to thank J. Leifer and J. Palsberg for comments. The first author was supported by a Royal Society University Research Fellowship and by EPSRC grant GR/L 62290 *Calculi for Interactive Systems: Theory and Experiment*. The second author did part of this work in the Object System Group at the University of Geneva.

References

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, pages 147–160, New York, NY, USA, 1999. ACM Press.

[AFG98] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. In *LICS 98 (Indiana)*, pages 105–116. IEEE, Computer Society Press, July 1998.

[BHR84] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sofia-Antipolis, May 1992.

[BS95] Michele Boreale and Davide Sangiorgi. A fully abstract semantics for causality in the pi-calculus. In E. W. Mayr and C. Puech, editors, *Proceedings of STACS'95*, volume 900 of *Lecture Notes in Computer Science*, pages 243–254. Springer-Verlag, 1995.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS), ETAPS'98, LNCS 1378*, pages 140–155, March 1998.

[DP95] Pierpaolo Degano and Corrado Priami. Causality for mobile processes. In Zoltán Fülöp and Ferenc Gécseg, editors, *Proceedings of ICALP '95*, volume 944 of *Lecture Notes in Computer Science*, pages 660–671. Springer-Verlag, 1995.

[FBF99] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, Berkeley, California, May 1999.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of CONCUR '96. LNCS 1119*, pages 406–421. Springer-Verlag, August 1996.

[GM99] Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, pages 250–261, New York, NY, USA, 1999. ACM Press.

[GRPA97] Douglas P. Ghormley, Steven H. Rodrigues, David Petrou, and Thomas E. Anderson. Interposition as an operating system extension mechanism. Technical Report CSD-96-920, University of California, Berkeley, April 9, 1997.

[GWTB96] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Sixth USENIX Security Symposium*, San Jose, California, July 1996.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th POPL*, January 1998.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings of ECOOP '91, LNCS 512*, pages 133–147, July 1991.

[Jon99] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programing: Security Issues for Mobile and Distributed Objects*. Springer Verlag, 1999.

[KML96] Myong H. Kang, Ira S. Moskowitz, and Daniel C. Lee. A network pump. *IEEE Transactions on Software Engineering*, 22(5):329–338, May 1996.

[LR98] Xavier Leroy and François Rouaix. Security properties of typed applets. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 391–403, San Diego, California, 19–21 January 1998.

[McL94] J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley & Sons, 1994.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I + II. *Information and Computation*, 100(1):1–77, 1992.

[NL98] G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 61–91. SV, 1998.

[PØ97] Jens Palsberg and Peter Ørbæk. Trust in the lambda-calculus. *Journal of Functional Programming*, 7(6):557–591, November 1997.

[PT99] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1999.

[RG99] A.W. Roscoe and M.H. Goldsmith. What is intransitive noninterference? In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW-12)*, Mordano, Italy, June 1999.

[RH98] James Riely and Matthew Hennessy. A typed language for distributed mobile processes. In *Proceedings of the 25th POPL*, January 1998.

[Sew98] Peter Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proceedings of ICALP '98, LNCS 1443*, pages 695–706, 1998.

[Sew99] Peter Sewell. A brief introduction to applied π, January 1999. Lecture notes for the Mathfit Instructional Meeting on Recent Advances in Semantics and Types for Concurrency: Theory and Practice, July 1998. Available from `http://www.cl.cam.ac.uk/users/pes20/`.

[SS99] Andrei Sabelfeld and David Sands. A PER model of secure information flow in sequential programs. In *Proceedings of European Symposium on Programming*, Amsterdam, Netherlands, March 1999.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, 19–21 January 1998.

[SV99a] Peter Sewell and Jan Vitek. Secure composition of insecure components. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW-12)*, Mordano, Italy, June 1999.

[SV99b] Peter Sewell and Jan Vitek. Secure composition of insecure components. Trusted objects, Centre Universitaire d'Informatique, University of Geneva, July 1999. Also available as University of Cambridge TR 463.

[SV99c] Peter Sewell and Jan Vitek. Secure composition of untrusted code: Wrappers and causality types. Technical Report 478, Computer Laboratory, University of Cambridge, November 1999.

[SWP99] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: a two-level architecture. In *Internet Programming Languages, LNCS 1686*. Springer-Verlag, October 1999.

[VC98] Jan Vitek and Guiseppe Castagna. Towards a calculus of mobile computations. In *Workshop on Internet Programming Languages, Chicago*, May 1998.

[VD98] Jose-Luis Vivas and Mads Dam. From higher-order pi-calculus to pi-calculus in the presence of static operators. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR '98: Concurrency Theory (9th International Conference, Nice, France)*, volume 1466 of *lncs*, pages 115–130. sv, September 1998.

[VS98] Dennis Volpano and Geoffrey Smith. Confinement properties for programming languages. *SIGACT News*, 29(3):33–42, September 1998.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):1–21, 1996.

[WBDF97] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. In *Proceedings of the 16th Symposium on Operating System Principles*, 1997.

[WN95] G. Winskel and M. Nielsen. Models for concurrency. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science*, volume IV, pages 1–148. Oxford University Press, 1995.

[ZGM99] Steve Zdancewic, Dan Grossman, and Greg Morrisett. Principals in programming languages: A syntactic proof technique. In *International Conference on Functional Programming*, Paris, France, September 1999.