Secure Composition of Insecure Components

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

Jan Vitek
Object Systems Group,
Université de Genève,
Switzerland
Jan.Vitek@cui.unige.ch

Abstract

Software systems are becoming heterogeneous: instead of a small number of large programs from well-established sources, a user's desktop may now consist of many smaller components that interact in intricate ways. Some components will be downloaded from the network from sources that are only partially trusted. A user would like to know that a number of security properties hold, e.g. that personal data is not leaked to the net, but it is typically infeasible to verify that such components are well-behaved. Instead, they must be executed in a secure environment, or wrapper, *that provides fine-grain control of the allowable interactions between them, and between components and other system resources.*

In this paper we study such wrappers, focusing on how they can be expressed in a way that enables their security properties to be stated and proved rigorously. We introduce a model programming language, the box-π calculus, that supports composition of software components and the enforcement of security policies. Several example wrappers are expressed using the calculus; we explore the delicate security properties they guarantee.

1 Introduction

Software systems are evolving. Increasingly, monolithic applications are being replaced with assemblages of software components coming from different sources. Instead of a small number of large programs from well-established suppliers, nowadays a user's desktop is made up of many smaller applications and software modules that interact in intricate ways to carry out a variety of information processing tasks. Moreover, whereas it used to be that a software base was fairly static and often controlled by a system administrator, it is now easy to download code from the network; technologies such as Java even allow an application program to be extended with new components while the program is running.

In such fluid operating environments, traditional security mechanisms and policies appear almost irrelevant. While passwords and access control mechanisms are adequate to protect the integrity of the computer system as whole, they utterly fail to address the issue of protecting the user from downloaded code being run from her account [19, 13, 27]. Approaches such as the Java sandbox that promise security by isolation are not satisfactory either: components can interact freely or not at all [35, 14]. What is needed is much finer-grained protection mechanisms that take into account the interconnection of software components and the specific security requirements of individual users.

We give a small motivating example (based on a true story) involving a fictional character, Karen, performing some financial computation. To manage her accounts she downloads a software package called *Quickest* from a company Q. Karen does not want any information about her to be leaked without her consent, so she would like to run Quickest in an environment that does not allow it access to the Internet (she has observed that it sometimes uploads information – presumably for marketing purposes – to Q). On the other hand she often needs stock quotes, for which she must allow net access. At present she runs two instances of Quickest, one on an isolated PC, with her financial records, and one connected, used to obtain stock quotes. She transfers data from the second to the first only on floppy disc, thereby manually ensuring that no information flows in the converse direction.

Karen would like to dispose of the isolated PC, using a software solution to prevent her personal data being leaked to the net. Now, Quickest is a large piece of commercial software that was not programmed by Karen. The source code is not available to her and its internal behaviour is complex and inaccessible; ensuring the desired properties by program analysis will not be feasible. Instead she must run the two copies of the package in secure software environments that allow control of the information flow between them and between each package and the net.

More generally, she will wish to run many packages, each trusted in different ways, and will want to be able to dynamically control the interactions between them and between these packages and other resources – the net, regions of the local disc, the terminal, audio and video capture devices etc. In some cases she will wish to log the data sent from one to another; in others she will wish to limit the allowed bandwidth (e.g. to disallow audio and video channels). In general her notion of what data is to be considered "sensitive" is likely to be context dependent. In a Web browser, she may choose to consider her e-mail address as a secret that should be protected from broadcast to junk mail lists, while the same e-mail will not be treated specially in her text editor.

While it is not feasible to analyse or modify large third-party software packages, it *is* possible to intercept the communications between a package and the other parts of the system, interposing code at the boundaries of the different software components [20, 11, 7, 13]. It is thus possible to monitor or control the operations that these components are able to invoke, and the data that is exchanged between them. We call a code fragment that encapsulates untrusted components a *security wrapper* or *wrapper* for short.

Clearly the task of writing wrappers should not be left solely to the end-user. Rather we envision wrappers as reusable software components, users should then only have to pick the most appropriate wrappers, customize them with some parameters and install them. All of this process should be dynamic: wrappers must be no harder to add to a running system than new applications. A user will require a clear description of the security properties that a wrapper guarantees. Moreover, wrappers should compose with a clear notion of which properties are preserved.

The goal of this work is to study such secure environments, focusing on how they can be expressed in a way that enables their security properties to be stated and proved rigorously. It appears that there is a wide range of rather delicate properties, making hard for designers to develop sufficiently clear intuitions without such rigour. Moreover the wrappers, although critical, may be rather small pieces of software, making it feasible to prove properties about them, or about mild idealisations.

To express and reason about wrappers we require a small programming language, with a well-defined semantics, that allows the composition of software components to be expressed straightforwardly and also supports the enforcement of security policies. Such a language, the box-π calculus, is introduced in §2. We begin with a simple example, a wrapper $\mathcal{W}_1$ written in the calculus. It encapsulates a single component and controls its interactions with the environment, limiting them to two channels in and out. $\mathcal{W}_1$ is written as a unary context:

$$\begin{array}{lll} \mathcal{W}_1[\_] & \stackrel{\mathit{def}}{=} & (\boldsymbol{\nu}\, a)\big(\quad a[\_] \\ & & \qquad\quad \mid \, !\, in^{\uparrow} y.\overline{in}^{a} y \\ & & \qquad\quad \mid \, !\, out^{a} y.\overline{out}^{\uparrow} y\big) \end{array}$$

This creates a box with a new name a, installing in parallel with it two forwarders – one that receives messages from the environment on channel in and sends them to the wrapped program, and one that receives messages from the wrapped program on channel out and sends them to the environment. An arbitrary program P (possibly malicious) can be wrapped to give $\mathcal{W}_1[P]$; the design of the calculus and of $\mathcal{W}_1$ ensures that no matter how P behaves the wrapped program $\mathcal{W}_1[P]$ can only interact with its environment on the two channels in and out. This could be achieved simply by forbidding *all* interaction between P and the outside world, a rather unsatisfactory wrapper — $\mathcal{W}_1$ is also *honest*, in that it faithfully forwards messages on in and out. These informal properties are made precise in Propositions 2 and 5 below. We also discuss the sense in which wrapping a *well-behaved* P has no effect on its behaviour. $\mathcal{W}_1$ is atypical in that it has no behaviour except the forwarding of legitimate messages – other reasonable unary wrappers may perform some kind of logging, or have a control interface for the wrapper. The honesty property that should hold for any reasonable wrapper is therefore somewhat delicate; to state it (and our other security properties) we make extensive use of a labelled transition semantics for the calculus.

The wrapper $\mathcal{W}_1$ controls interaction between a single component and its environment. Our second main example goes further towards solving Karen's problem, allowing control of the interaction between components. $\mathcal{W}_2$ (defined in §3) is a binary wrapper that encapsulates two components P and Q as $\mathcal{W}_2[P, Q]$, allowing each to interact with the environment in a limited way but also allowing information to flow from P to Q (but not vice versa) along a directed communication channel. Making this precise is the subject of §5.

Both $\mathcal{W}_1$ and $\mathcal{W}_2$ are chosen to be as simple as possible, in particular with fixed interfaces for components to interact with each other and with the environment. Generalising this to arbitrary interfaces and to wrappers taking any number of components should be straightforward but complicates the notation; other generalisations are discussed in the conclusion.

Overview We begin in the next section (§2) by introducing the calculus and giving its operational semantics. A number of wrappers are defined in §3, including one which logs traffic. The basic properties of honesty and well-behaviour are introduced in §4. Information flows between wrapped components are studied in §5, then we conclude in §6 with discussion of related and future work. This paper describes work in progress – Sections 4 and 5 contain a number of conjectures which are yet to be proved, but which we hope will stimulate discussion.

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 non-determinism. 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 [24]. Related calculi have been used by a number of authors, e.g. in [2, 4, 6, 9, 10, 12, 17, 16, 28, 30, 31, 33, 34, 36, 37]. A brief overview of the design space can be found in [32]; here we highlight the main design choices for box-π, deferring comparison with related work to §6.

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 [18, 8, 5]. 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.

To π we must add primitives 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, 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* [36]. Communication is therefore allowed only between a box and its parent, or within the process running in a particular box. In particular, two sibling boxes cannot interact without the assistance of their parent. To enable a box to interact with a particular child, boxes are named, analogously to π channel names. The security properties of our wrappers depend on the ability to create fresh box names.

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. 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 lower-case roman letters n, m, x, y, z 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 upon reception. 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
		$\langle p_1 \mathinner{..} p_k \rangle$	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
		$\ldots$	

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

$$
\begin{array}{lll}
P & ::= & \ldots \\
 & & \overline{x}^{o}v \qquad \text{output } v \text{ on channel } x \text{ to } o \\
 & & x^{\iota}p.P \qquad \text{input on channel } x \text{ from } \iota \\
 & & !\,x^{\iota}p.P \qquad \text{replicated input} \\
 & & \ldots
\end{array}
$$

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.

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.

$$
\begin{array}{lll}
P & ::= & \ldots \\
 & & (\boldsymbol{\nu}\, x)P \qquad \text{new name creation}
\end{array}
$$

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 Reduction

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 write $\{^{v}/_{p}\}P$ for the result of substituting appropriate parts of the value v for the names of the pattern p in P. Note that this may be undefined, either because the value does not match the pattern or because the syntax does not allow arbitrary values in all the places where free names can occur. We define structural congruence $\equiv$ as the least congruence relation such that the axioms below hold. This allows the parts of a redex to be brought syntactically adjacent.

$$
\begin{array}{rcll}
P \mid 0 & \equiv & P & \\
P \mid Q & \equiv & Q \mid P & \\
(P \mid Q) \mid R & \equiv & P \mid (Q \mid R) & \\
(\boldsymbol{\nu}\, x)(\boldsymbol{\nu}\, y)P & \equiv & (\boldsymbol{\nu}\, y)(\boldsymbol{\nu}\, x)P & \\
(\boldsymbol{\nu}\, x)(P \mid Q) & \equiv & P \mid (\boldsymbol{\nu}\, x)Q & x \notin \mathrm{fn}(P) \\
(\boldsymbol{\nu}\, x)n[P] & \equiv & n[(\boldsymbol{\nu}\, x)P] & x \neq n
\end{array}
$$

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.

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

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 thus take two reduction steps, for example in the following upwards and downwards communications.

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

This removes the need for 3-way synchronisations between a box, an output and an input (as in [36]), simplifying both the semantics and the implementation model.

2.3 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 an SOS. The *labels* are

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

where γ ranges over all output tags except $\overline{\uparrow}$. 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

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

$$\frac{}{\overline{x}^{o} v \xrightarrow{\overline{x}^{o} v} 0}\ \text{(Out)} \qquad \frac{}{x^{\iota} p.P \xrightarrow{x^{\iota} v} \{^{v}/_{p}\}P}\ \text{(In)} \qquad \frac{}{!\, x^{\iota} p.P \xrightarrow{x^{\iota} v}\, !\, x^{\iota} p.P \mid \{^{v}/_{p}\}P}\ \text{(Repl)}$$

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

$$\frac{A \vdash P \xrightarrow{\overline{x}^{\uparrow} v} P'}{A \vdash n[P] \xrightarrow{\tau} (\boldsymbol{\nu}\, \mathrm{fn}(x,v) - A)(\overline{x}^{\overline{n}} v \mid n[P'])}\ \text{(Box-1)} \qquad \frac{}{n[P] \xrightarrow{x^{\overline{n}} v} n[\overline{x}^{\overline{\uparrow}} v \mid P]}\ \text{(Box-2)} \qquad \frac{P \xrightarrow{\tau} P'}{n[P] \xrightarrow{\tau} n[P']}\ \text{(Box-3)}$$

$$\frac{A, x \vdash P \xrightarrow{\ell} P'}{A \vdash (\boldsymbol{\nu}\, x)P \xrightarrow{\ell} (\boldsymbol{\nu}\, x)P'}\ \text{(Res-1)} \qquad \frac{A, x \vdash P \xrightarrow{\overline{y}^{o} v} P'}{A \vdash (\boldsymbol{\nu}\, x)P \xrightarrow{\overline{y}^{o} v} P'}\ \text{(Res-2)} \qquad \frac{P \xrightarrow{\ell} P' \quad P' \equiv P''}{P \xrightarrow{\ell} P''}\ \text{(Struct Right)}$$

The (Res-1) rule is subject to $x \notin \mathrm{fn}(\ell)$, the (Res-2) rule is subject to $x \in \mathrm{fn}(v) - \mathrm{fn}(y, o)$ if $\neg \mathrm{mv}(o)$ and to $x \in \mathrm{fn}(y, v) - \mathrm{fn}(o)$ otherwise. The indexing $A \vdash$ has beeen elided in rules where it is not involved in any interesting way. In all rules with conclusion of the form $A \vdash P \xrightarrow{\ell} Q$ there is an implicit side condition $\mathrm{fn}(P) \subseteq A$. In the (In) and (Repl) axioms there is an implicit side condition that $\{^{v}/_{p}\}P$ is well-defined. Symmetric versions of (Par) and (Comm) are elided.

Figure 1. Box-π Labelled Transitions

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 will synchronise in the pairs given:

$$\begin{array}{ll} \overline{x}^{\star} v & x^{\star} v \\ \overline{x}^{\overline{n}} v & x^{n} v \\ \overline{x}^{\overline{\uparrow}} v & x^{\uparrow} v \\ \overline{x}^{n} v & x^{\overline{n}} v \end{array}$$

The labelled transition relation has the form

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

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 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 1. 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. For the subcalculus without new-binding the labelled transition rules are straightforward — instances of the reduction rule (Red Up) correspond to uses of (Box-1), (Out), and (Par); instances of (Red Down) correspond to uses of (Comm), (Out), and (Box-2); instances of (Red Comm) correspond to uses of (Comm), (Out), and (In). The addition of new-binding introduces several subtleties, some inherited from the π-calculus and some related to scope extrusion and intrusion across box boundaries. We discuss the latter briefly.

The (Red Down) rule involves synchronisation on the box name n but *not* on the channel name x — there are reductions such as

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

in which a new-bound name enters a box boundary. To correctly match this with a τ-transition the side-condition for (Res-2) for labels with output tag n requires the bound name to occur either in channel or value position, and the (Comm) rule reintroduces the x binder on the right hand side.

Similarly, the (Red Up) rule allows new-bound names in channel position to exit a box boundary, for example in

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

The (Res-2) condition for output tag $\uparrow$ again requires the bound name to occur either in channel or value position, here the (Box-1) rule reintroduces the x binder on the right hand side.

Reductions generated by (Red Comm) involve synchronisation both on the tags and on the channel name. The (Res-2) condition for output tags $\star$, $\overline{\uparrow}$ and $\overline{n}$ is analogous to the standard π-calculus (Open) rule; requiring the bound name to occur in the value but not in the tag or channel. The (Comm) rule for these output tags is analogous to the standard π rule — in particular, here it is guaranteed that $x \in A$ (see Lemma 10).

Some auxiliary notation is useful. For a sequence of labels $\ell_1 \dots \ell_k$ we write

$$A \vdash P_1 \xrightarrow{\ell_1} \dots \xrightarrow{\ell_k} P_{k+1}$$

to mean $\exists P_2, \dots, P_k \ . \ \forall i \in 1..k \ . \ A_i \vdash P_i \xrightarrow{\ell_i} P_{i+1}$, where $A_i = A \cup \bigcup_{j \in 1..i} \mathrm{fn}(\ell_j)$. If $\ell \neq \tau$ we write $A \vdash P \stackrel{\hat{\ell}}{\Longrightarrow} P'$ for $A \vdash P \xrightarrow{\tau}^* \xrightarrow{\ell} \xrightarrow{\tau}^* P'$; if $\ell = \tau$ then $A \vdash P \stackrel{\hat{\ell}}{\Longrightarrow} P'$ is defined as $A \vdash P \xrightarrow{\tau}^* P'$.

The two semantics coincide in the following sense.

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

This give confidence that the labelled semantics carries enough information. The proof is somewhat delicate — it is sketched in Appendix A; full details can be found in the forthcoming technical report.

2.4 Bisimulation

The statements of some relationships between the behaviour of a wrapped and an unwrapped program require an operational equivalence relation. As box-π is asynchronous, an appropriate notion can be based on the *weak asynchronous bisimulation* of [5]. Consider a family S of relations indexed by finite sets of names such that each S_A is a symmetric relation over $\{ P \mid \mathrm{fn}(P) \subseteq A \}$. Say S is a *weak asynchronous bisimulation* if

- $P \ S_A \ Q$, $A \vdash P \xrightarrow{\ell} P'$ and ℓ is an output or τ transition imply $\exists Q' \ . \ A \vdash Q \stackrel{\hat{\ell}}{\Longrightarrow} Q' \wedge P' \ S_{A \cup \mathrm{fn}(\ell)} \ Q'$, and
- $P \ S_A \ Q$, $A \vdash P \xrightarrow{x^\gamma v} P'$ imply either $\exists Q' \ . \ A \vdash Q \stackrel{x^\gamma v}{\Longrightarrow} Q' \wedge P' \ S_{A \cup \mathrm{fn}(x^\gamma v)} \ Q'$ or $\exists Q' \ . \ A \vdash Q \Longrightarrow Q' \wedge P' \ S_{A \cup \mathrm{fn}(x^\gamma v)} \ (Q' \mid \overline{x}^\gamma v)$.

We write $\approx$ for the union of all weak asynchronous bisimulations. (This definition has not been thoroughly tested – in particular, it has not been proved to be a congruence.)

3 Security Wrappers

This section gives three example wrappers. The first encapsulates a single component, restricting its interactions with the outside world to communications obeying a certain protocol. The second is similar, but also writes a log of all such communications. The third wrapper encapsulates two components, allowing each to interact with the outside world in a limited way but also allowing information to flow from the first to the second (but not vice versa).

A wrapper design must be in the context of some fixed protocol which components should use for communication with their environment and with each other. For the first two wrappers we fix two channel names, in and out, for components to receive and send messages respectively. Moreover, we assume that components will always be executed within some box and should be communicating with the parent box. A trivial component that receives values v and then copies pairs $\langle v\, v \rangle$ to the output would be written as

$$! \, in^\uparrow y . \overline{out}^\uparrow \langle y\, y \rangle$$

A malicious component might also write data to another illicit output channel available in the environment, e.g.

$$! \, in^\uparrow y . (\overline{net}^\uparrow y \mid \overline{out}^\uparrow \langle y\, y \rangle)$$

or eavesdrop on communications between other parts of the system, e.g.

$$! \, c^\star y . (\overline{net}^\uparrow c \mid \overline{c}^\star y)$$

We can express whether a component obeys the protocol in terms of the labelled transition semantics – say P is *well-behaved* for a unary wrapper iff whenever $A \vdash P \xrightarrow{l_1 .. l_k} Q$ then the l_j are of the form $in^\uparrow v$, $\overline{out}^\uparrow v$, or τ.

A Filtering Wrapper A filter is a wrapper that simply restricts the communication abilities of a process. We consider a static filter that allows interaction on two channels in and out only.

$$\begin{array}{rcll} \mathcal{W}_1[\_] & \stackrel{def}{=} & (\boldsymbol{\nu}\, a)(& a[\_] \\ & & & \mid \, ! \, in^\uparrow y . \overline{in}^a y \\ & & & \mid \, ! \, out^a y . \overline{out}^\uparrow y \ ) \end{array}$$

$\mathcal{W}_1$ executes its component within a freshly-named box, installing forwarders to move legitimate messages across the boundary. Note that this and further wrappers are non-binding contexts – equivalently, we assume wherever we apply $\mathcal{W}_1$ to a process P that the new-bound a does not occur free in P (in an implementation this could be ensured either probabilistically or with a linear-time scan of P). Irrespective of the behaviour of P, $\mathcal{W}_1[P]$ does obey the protocol – this can be stated clearly using the labelled transition semantics:

Proposition 2 *For any program P with $a \notin \mathrm{fn}(P)$, if $A \vdash \mathcal{W}_1[P] \xrightarrow{l_1 .. l_k} Q$ then the l_j are of the form $in^\uparrow v$, $\overline{out}^\uparrow v$, or τ.*

The proof is via an explicit characterisation of the states reachable by labelled transitions of $\mathcal{W}_1[P]$. A sketch of this, and of the other properties of $\mathcal{W}_1$, can be found in Appendix B; full details can be found in the forthcoming technical report. We say a unary wrapper with this property is *pure*.

The Logging Wrapper The filter can be extended to maintain a log of all communications of a process, sending copies on a channel *log* to the environment:

$$\begin{array}{rcl} \mathcal{L}[\_] & \stackrel{def}{=} & (\boldsymbol{\nu}\, a)\big(\quad a[\_] \\ & & \qquad \mid \,!\, in^{\uparrow} y.(\overline{log}^{\uparrow} y \mid \overline{in}^{a} y) \\ & & \qquad \mid \,!\, out^{a} y.(\overline{log}^{\uparrow} y \mid \overline{out}^{\uparrow} y)\ \big) \end{array}$$

A wrapped program $\mathcal{L}[P]$ again can interact only in limited ways.

Proposition 3 *For any program P with $a \notin \mathrm{fn}(P)$, if $A \vdash \mathcal{L}[P] \xrightarrow{l_1..l_n} Q$ then the l_j are of the form $in^{\uparrow} v$, $\overline{out}^{\uparrow} v$, $\overline{log}^{\uparrow} v$, or τ.*

A Pipeline Wrapper A pipeline wrapper allows a controlled flow of information between two components. We give a binary wrapper $\mathcal{W}_2$ that takes two processes. In an execution of $\mathcal{W}_2[Q_1, Q_2]$ the two wrapped processes Q_i can interact with the environment as before, on channels in_i and out_i. In addition, Q_1 can send messages to Q_2 on a channel mid. The pipeline implemented here is unordered.

$$\begin{array}{rcl} \mathcal{W}_2[\_1, \_2] & \stackrel{def}{=} & (\boldsymbol{\nu}\, a_1, a_2)\big(\quad a_1[\_1] \mid a_2[\_2] \\ & & \qquad \mid \,!\, {in_1}^{\uparrow} y.\overline{in_1}^{a_1} y \\ & & \qquad \mid \,!\, {in_2}^{\uparrow} y.\overline{in_2}^{a_2} y \\ & & \qquad \mid \,!\, {out_1}^{a_1} y.\overline{out_1}^{\uparrow} y \\ & & \qquad \mid \,!\, {out_2}^{a_2} y.\overline{out_2}^{\uparrow} y \\ & & \qquad \mid \,!\, mid^{a_1} y.\overline{mid}^{a_2} y\ \big) \end{array}$$

As before $\mathcal{W}_2$ is a non-binding context – we assume, wherever we apply it to two processes P_1, P_2, that $\{a_1, a_2\} \cap \mathrm{fn}(P_1, P_2) = \emptyset$. Say a binary wrapper $\mathcal{C}$ is pure iff for any programs P_1, P_2, (satisfying the appropriate free name condition – for $\mathcal{W}_2$ that with $\{a_1, a_2\} \cap \mathrm{fn}(P_1, P_2) = \emptyset$), if $A \vdash \mathcal{C}[P_1, P_2] \xrightarrow{l_1..l_k} Q$ then the l_j are of the form ${in_i}^{\uparrow} v$, $\overline{out_i}^{\uparrow} v$, or τ.

Proposition 4 *$\mathcal{W}_2$ is pure.*

For an example of a blocked attempt by the second process to send a value to the first, suppose $P_2 = \overline{mid}^{\uparrow} v$. We have

$$\begin{array}{l} \mathcal{W}_2[P_1, \overline{mid}^{\uparrow} v] = (\boldsymbol{\nu}\, a_1, a_2)\big(a_1[P_1] \mid a_2[\overline{mid}^{\uparrow} v] \mid R\big) \\ \qquad \rightarrow (\boldsymbol{\nu}\, a_1, a_2)\big(a_1[P_1] \mid a_2[0] \mid \overline{mid}^{a_2} v \mid R\big) \end{array}$$

where R is the parallel composition of forwarders. The output $\overline{mid}^{a_2} v$ in the final state cannot interact further – not with the environment, as a_2 is restricted, and not with the forwarder $!\, mid^{a_1} y.\overline{mid}^{a_2} y$, as $a_1 \neq a_2$.

These wrappers all assume a rather simple fixed protocol. It would be straightforward to generalise to arbitrary sets of channels instead of in, out and mid. It would also be straightforward to allow n-ary wrappers, encapsulating many components and allowing information to flow only on a given preorder between them. Other generalisations are discussed in the conclusion.

4 Honesty and Composition

The properties of wrappers stated in the previous section are very weak. For example, the unary wrapper

$$\mathcal{C}[\_] \stackrel{def}{=} 0$$

is also pure, but is useless. In this section we identify the class of *honest* wrappers that are guaranteed to forward legitimate messages. This gives the authors of components a clear statement of (some of) the properties of the environment that can be relied upon.

An initial attempt might be to take $\mathcal{W}_1$ as a specification, defining a unary wrapper $\mathcal{C}$ to be honest iff for any program P the processes $\mathcal{C}[P]$ and $\mathcal{W}_1[P]$ are operationally equivalent. This is unsatisfactory – it rules out wrappers such as $\mathcal{L}$, and it does not give a very clear statement of the properties that may be assumed of an honest wrapper.

A better attempt might be to say that a unary wrapper $\mathcal{C}$ is honest iff for any well-behaved P the processes $\mathcal{C}[P]$ and P are operationally equivalent. This would be unsatisfactory in two ways. Firstly, some intuitively sound wrappers have additional interactions with the environment – e.g. the logging outputs of $\mathcal{L}$ – and so would not be considered honest by this definition. Secondly, this definition would not constrain the behaviour of wrappers for non-well-behaved P at all – if a component P attempted, in error, a single illicit communication then $\mathcal{C}[P]$ might behave arbitrarily.

To address these points we give explicit definitions of honesty, first for unary wrappers and then for binary, in the style of weak asynchronous bisimulation. Consider a family R indexed by finite sets of names such that each R_A is a relation over $\{\, P \mid \mathrm{fn}(P) \subseteq A \,\}$. Say R is an *h-bisimulation* if, whenever $C\ R_A\ Q$ then:

1. if $A \vdash C \xrightarrow{\ell} C'$ for $\ell = \overline{out}^{\uparrow} v, \tau$ then $A \vdash Q \stackrel{\hat{\ell}}{\Longrightarrow} Q' \wedge C'\ R_{A \cup \mathrm{fn}(\ell)}\ Q'$
2. if $A \vdash C \xrightarrow{in^{\uparrow} v} C'$ then either $A \vdash Q \stackrel{in^{\uparrow} v}{\Longrightarrow} Q'$ and $C'\ R_{A \cup \mathrm{fn}(in, v)}\ Q'$ or $A \vdash Q \Longrightarrow Q'$ and $C'\ R_{A \cup \mathrm{fn}(in, v)}\ Q' \mid \overline{in}^{\uparrow} v$

3. if $A \vdash C \xrightarrow{\ell} C'$ for any other label then $C' \; R_{A\cup\mathrm{fn}(\ell)} \; Q$

together with symmetric versions of clauses 1 and 2. Say a unary wrapper $\mathcal{C}$ is *honest* if for any program P (satisfying the appropriate free name condition) and any $A \supseteq \mathrm{fn}(\mathcal{C}[P])$ there is an h-bisimulation R with $\mathcal{C}[P] \; R_A \; P$.

Loosely, clauses 1, 2 and the symmetric versions ensure that legitimate communications and internal reductions must be weakly matched. Clause 3 ensures that if the wrapper performs some additional communication then this does not affect the state as seen by the wrapped process.

Proposition 5 *The unary wrappers $\mathcal{W}_1$ and $\mathcal{L}$ are honest.*

We give some examples of dishonest wrappers. Take

$$\mathcal{C}[\_] \quad \stackrel{def}{=} \quad (\boldsymbol{\nu}\, a)a[\_]$$

This is not honest – a transition $A \vdash P \xrightarrow{\overline{out}^{\uparrow} v} P'$ cannot be matched by $\mathcal{C}[P]$, violating the symmetric version of clause 1. Now consider

$$\mathcal{C}[\_] \quad \stackrel{def}{=} \quad \_$$

This wrapper is also dishonest as $\mathcal{C}[P]$ can perform actions not in the protocol that essentially affect the state of P. For example, take $P = x^{\star}y.\overline{out}^{\uparrow}\langle\rangle$. Suppose $\mathcal{C}[P] \; R_A \; P$ for an h-bisimulation R. We have $A \vdash \mathcal{C}[P] \xrightarrow{x^{\star}\langle\rangle} \overline{out}^{\uparrow}\langle\rangle$ so by clause 3 $\overline{out}^{\uparrow}\langle\rangle \; R_A \; P$, but then clause 1 cannot hold – the left hand side can perform an $\overline{out}^{\uparrow}\langle\rangle$ transition that cannot be matched be the right hand side.

Composition of Wrappers The protocol for communication between a component and a unary wrapper is designed so that wrappers may be nested. We conjecture that the composition of any honest unary wrappers is honest.

Conjecture 6 *If $\mathcal{C}_1$ and $\mathcal{C}_2$ are honest unary wrappers then $\mathcal{C}_1 \circ \mathcal{C}_2$ is honest.*

Analogous results for non-unary wrappers would require wrappers with more complex interfaces so that the input, output and mid channels could be connected correctly.

A desirable property of a pure wrapper is that it should not affect the behaviour of any well-behaved component — one might expect for any pure and honest $\mathcal{C}$ and well-behaved P that $\mathcal{C}[P] \approx_A P$ (where $A \supseteq \mathrm{fn}(\mathcal{C}[P])$). Unfortunately this does not hold, even for $\mathcal{W}_1$, as the wrapper can make input transitions that cannot be matched. One can check $\mathcal{W}_1[0] \not\approx_A 0$, yet 0 is well-behaved. In practice one would expect the environment of a wrapper to not be able to detect these inputs, but to make this precise would require an operational equivalence relativised to such 'well-behaved' environments.

A simpler property would be that multiple wrappings have no effect. We conjecture that $\mathcal{W}_1$ is idempotent, i.e. that $\mathcal{W}_1[\mathcal{W}_1[P]]$ and $\mathcal{W}_1[P]$ have the same behaviour (up to weak asynchronous bisimulation):

Conjecture 7 *For any program P with $a \notin \mathrm{fn}(P)$ and $A \supseteq \mathrm{fn}(\mathcal{W}_1[P])$ we have $\mathcal{W}_1[P] \approx_A \mathcal{W}_1[\mathcal{W}_1[P]]$.*

Honesty for Binary Wrappers must take into account the mid communication. Consider a family R indexed by finite sets of names such that each R_A is a relation between terms and pairs of terms, all with free names contained in A. Say R is a *binary h-bisimulation* if, whenever $C \; R_A \; (Q_1, Q_2)$ the clauses below hold. The key difference with the unary definition is clause 7; the other clauses are routine, albeit notationally complex.

1. if $A \vdash C \xrightarrow{\overline{out_i}^{\uparrow} v} C'$ then $A \vdash Q_i \stackrel{\overline{out_i}^{\uparrow} v}{\Longrightarrow} Q_i'$, $A \vdash Q_{3-i} \Longrightarrow Q_{3-i}'$ and $C' \; R_{A\cup\mathrm{fn}(v)} \; (Q_1', Q_2')$.
2. if $A \vdash C \xrightarrow{in_i{}^{\uparrow} v} C'$ then $A \vdash Q_{3-i} \Longrightarrow Q_{3-i}'$ and either $A \vdash Q_i \stackrel{in_i{}^{\uparrow} v}{\Longrightarrow} Q_i' \wedge C' \; R_{A\cup\mathrm{fn}(v)} \; (Q_1', Q_2')$ or $A \vdash Q_i \Longrightarrow Q_i'' \wedge C' \; R_{A\cup\mathrm{fn}(v)} \; (Q_1', Q_2')$, where $Q_i' = Q_i'' \mid \overline{in}^{\uparrow} v$.
3. if $A \vdash C \xrightarrow{\tau} C'$ then $A \vdash Q_1 \Longrightarrow Q_1'$, $A \vdash Q_2 \Longrightarrow Q_2'$ and $C' \; R_A \; (Q_1', Q_2')$.
4. if $A \vdash C \xrightarrow{\ell} C'$ for any other label then $C' \; R_{A\cup\mathrm{fn}(\ell)} \; (Q_1, Q_2)$
5. if $A \vdash Q_i \xrightarrow{\ell} Q_i'$ for $\ell = \overline{out_i}^{\uparrow} v, \tau$ then $A \vdash C \stackrel{\hat{\ell}}{\Longrightarrow} C'$, and $C' \; R_{A\cup\mathrm{fn}(\ell)} \; (Q_1', Q_2')$, where $Q_{3-i}' = Q_{3-i}$.
6. if $A \vdash Q_i \xrightarrow{in_i{}^{\uparrow} v} Q_i'$ then either $A \vdash C \stackrel{in_i{}^{\uparrow} v}{\Longrightarrow} C' \wedge C' \; R_{A\cup\mathrm{fn}(v)} \; (Q_1', Q_2')$ or $A \vdash C \Longrightarrow C' \wedge C' \mid \overline{in}^{\uparrow} v \; R_{A\cup\mathrm{fn}(v)} \; (Q_1', Q_2')$, where $Q_{3-i}' = Q_{3-i}$.
7. if $A \vdash Q_1 \xrightarrow{\overline{mid}^{\uparrow} v} Q_1'$ then $A \vdash C \Longrightarrow C' \wedge C' \; R_{A\cup\mathrm{fn}(v)} \; (Q_1', Q_2 \mid \overline{mid}^{\uparrow} v)$.

A binary wrapper $\mathcal{C}$ is honest if for all P_1, P_2 (satisfying the appropriate free name condition) and any $A \supseteq \mathrm{fn}(\mathcal{C}[P_1, P_2])$ there exists a binary h-bisimulation R with $\mathcal{C}[P_1, P_2] \; R_A \; (P_1, P_2)$.

Conjecture 8 *$\mathcal{W}_2$ is honest.*

5 Constrained Interaction Between Components

In our motivating example Karen required fine-grain control over the information flows between components – in the binary case, allowing unidirectional flow. By examining the code for $\mathcal{W}_2$ it is intuitively clear that it achieves this, preventing information flowing from Q to P within $\mathcal{W}_2[P, Q]$. When one comes to make this intuition precise, however, it becomes far from clear exactly what behavioural properties $\mathcal{W}_2$ guarantees that make it a satisfactory wrapper from the user's point of view (who should not have to examine the wrapper code). Honesty is one, but it does not prohibit bad flows. In this section we give a number of candidate properties, stating four precisely and the others informally. We conjecture that all are satisfied by $\mathcal{W}_2$ but that none are equivalent. None are entirely satisfactory; we hope to provoke discussion of exactly what guarantees should be desired by users and by component designers. For simplicity, only pure binary wrappers $\mathcal{C}$ are considered – recall that for a pure binary $\mathcal{C}$ the labelled transitions of $\mathcal{C}[P_1, P_2]$ will only be of the forms $in_i{}^{\uparrow}v$, $\overline{out_i}^{\uparrow}v$ and τ.

New-name directionality As we are using a calculus with creation of new names, we can test a wrapper by supplying a new name to the second component, on in_2, and observing whether it can ever be output by the first component on out_1. Say $\mathcal{C}$ is *directional for new names* if whenever

$$A \vdash \mathcal{C}[P_1, P_2] \xrightarrow{\ell_1} \ldots \xrightarrow{\ell_j} \xrightarrow{in_2{}^{\uparrow}u} \xrightarrow{\ell'_1} \ldots \xrightarrow{\ell'_k} \xrightarrow{\overline{out_1}^{\uparrow}u'} P$$

with $x \in \mathrm{fn}(u)$, but x is new, i.e. $x \notin A \cup \mathrm{fn}(\ell_1 \ldots \ell_j)$, and x is not subsequently input to the first component, i.e.

$$x \notin \bigcup_{i \in 1..k \wedge \ell'_i = in_1{}^{\uparrow}v} \mathrm{fn}(v)$$

then x is not output by the first component, i.e. $x \notin \mathrm{fn}(u')$. This property does not prevent all information flow, however – a variant of $\mathcal{W}_2$ containing a reverse-forwarder that only forwards particular values, such as

$$!\, mid^{a_2}y.\mathbf{if}\ y \in \{0, 1\}\ \mathbf{then}\ \overline{mid}^{a_1}y$$

could still satisfy it. (Here 0 and 1 are free names, which must therefore be in A.)

Note that a binary wrapper $\mathcal{C}$ is intended only to limit information flow *within* $\mathcal{C}[P_1, P_2]$. We do not wish to place any constraint on the environment of the wrapper, for example forbidding the environment to copy values received from out_2 to in_1. Such a restriction could only be imposed by draconian measures, e.g. by waiting for P_1 to terminate before starting P_2, that would not be acceptable to the desktop user. Many programs are essentially non-terminating; if they are executing concurrently then the user cannot be prevented from reading the output of one and copying it to the other. In many circumstances this should be explicitly supported by the desktop cut-and-paste, perhaps with a warning signal.

Permutation Our second property formalises the intuition that if no observable behaviour due to P_1 depends on the behaviour of P_2 then in any trace it should be possible to move the actions associated with P_1 before all actions associated with P_2. Say $\mathcal{C}$ has the *permutation property* if whenever

$$A \vdash \mathcal{C}[P_1, P_2] \xrightarrow{\ell_1} \ldots \xrightarrow{\ell_k} P$$

with $\ell_i \neq \tau$ there exists a permutation π of $\{1, \ldots, k\}$ such that

$$A \vdash \mathcal{C}[P_1, P_2] \xrightarrow{\ell_{\pi(1)}} \ldots \xrightarrow{\ell_{\pi(k)}} P$$

and no in_1 or out_1 transition occurs after any in_2 or out_2 transition in $\ell_{\pi(1)} \ldots \ell_{\pi(k)}$. For an example wrapper without this property, consider

$$\begin{array}{rcl} \mathcal{C}[\_1, \_2] & \stackrel{\mathit{def}}{=} & (\boldsymbol{\nu}\, a_1, a_2)\big(\, a_1[\_1] \mid a_2[\_2] \\ & & \quad \mid\, !\, in_2{}^{\uparrow}y.(\overline{in_2}^{a_2}y \mid\, !\, in_1{}^{\uparrow}y.\overline{in_1}^{a_1}y) \\ & & \quad \mid\, !\, out_1{}^{a_1}y.\overline{out_1}^{\uparrow}y \\ & & \quad \mid\, !\, out_2{}^{a_2}y.\overline{out_2}^{\uparrow}y \\ & & \quad \mid\, !\, mid^{a_1}y.\overline{mid}^{a_2}y\big) \end{array}$$

Here the in_1 messages are not forwarded until at least one in_2 input is received from the environment. Nonetheless, in some sense there is still no information flow from the second component to the first.

The new-name directionality and permutation properties are expressed purely in terms of the externally observable behaviour of $\mathcal{C}[P, Q]$ (in fact, they are properties of its trace set, a very extensional semantics). Note, however, that the intuitive statement that information does not flow from Q to P depends on an understanding of the internal computation of P and Q that is not present in the reduction or labelled transition relations (given only that $\mathcal{C}[P, Q] \rightarrow^{*} R$ there is no way to associate subterms of R with an 'origin' in $\mathcal{C}$, P or Q). Our next two properties involve a more intensional semantics in which output and input processes are tagged with sets of colours. The semantics propagates colours in interaction steps, thereby tracking the dependencies of reductions.

Coloured Reductions Take a set col of colours (disjoint from $\mathcal{N}$), and let c and d range over subsets of col. We define a coloured box-π calculus by annotating all outputs and inputs with sets of colours:

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

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 write cd for the union $\mathsf{c} \cup \mathsf{d}$. The reduction relation now takes the form $P \rightarrow_{\mathsf{c}} Q$, where P and Q are coloured terms and c is a set of colours indicating what this reduction depends upon. It is defined as follows, in which structural congruence is defined by the same axioms as before.

$$
\begin{array}{ll}
n[\mathsf{c}\!:\!\overline{x}^{\uparrow} v \mid Q] \rightarrow_{\mathsf{c}} \mathsf{c}\!:\!\overline{x}^{\overline{n}} v \mid n[Q] & \text{(C Red Up)} \\
\mathsf{c}\!:\!\overline{x}^{n} v \mid n[Q] \rightarrow_{\mathsf{c}} n[\mathsf{c}\!:\!\overline{x}^{\overline{\uparrow}} v \mid Q] & \text{(C Red Down)} \\
\mathsf{c}\!:\!\overline{x}^{\overline{\iota}} v \mid \mathsf{d}\!:\!x^{\iota} p.P \rightarrow_{\mathsf{cd}} \mathsf{cd} \bullet (\{^{v}/_{p}\}P) & \text{(C Red Comm)} \\
\mathsf{c}\!:\!\overline{x}^{\overline{\iota}} v \mid \mathsf{d}\!:\!\mathbf{!}\,x^{\iota} p.P \rightarrow_{\mathsf{cd}} & \\
\quad \mathsf{d}\!:\!\mathbf{!}\,x^{\iota} p.P \mid \mathsf{cd} \bullet (\{^{v}/_{p}\}P) & \text{(C Red Repl)} \\
P \rightarrow_{\mathsf{c}} Q \;\Rightarrow\; P \mid R \rightarrow_{\mathsf{c}} Q \mid R & \text{(C Red Par)} \\
P \rightarrow_{\mathsf{c}} Q \;\Rightarrow\; (\boldsymbol{\nu}\, x)P \rightarrow_{\mathsf{c}} (\boldsymbol{\nu}\, x)Q & \text{(C Red Res)} \\
P \rightarrow_{\mathsf{c}} Q \;\Rightarrow\; n[P] \rightarrow_{\mathsf{c}} n[Q] & \text{(C Red Box)} \\
P \equiv P' \rightarrow_{\mathsf{c}} Q' \equiv Q \;\Rightarrow\; P \rightarrow_{\mathsf{c}} Q & \text{(C Red Struct)}
\end{array}
$$

The coloured calculus has the same essential behaviour as the original calculus:

Proposition 9 *For any coloured P we have $|P| \rightarrow Q$ iff $\exists \mathsf{c}, P' \,.\, P \rightarrow_{\mathsf{c}} P' \wedge |P'| = Q$.*

Mediation We can now capture the intuition that all interaction between wrapped components should be mediated by the wrapper. We consider coloured reduction sequences of a wrapper $\mathcal{C}$ and two components P_1, P_2 from an initial state in which each is coloured differently. Let gr, bl and rd be distinct singleton subsets $\{\mathsf{green}\}$, $\{\mathsf{blue}\}$, $\{\mathsf{red}\}$ of col. Suppose

$$(\mathsf{gr} \circ \mathcal{C})\big[\mathsf{bl} \circ P_1,\, \mathsf{rd} \circ P_2\big] \mid \mathsf{bl} \circ I_1 \mid \mathsf{rd} \circ I_2 \rightarrow_{\mathsf{c}_1} \ldots \rightarrow_{\mathsf{c}_k} Q$$

where each I_i is a parallel composition of messages on in_i, i.e. of terms of the form $\overline{in_i}^{\overline{\uparrow}} v$. Say $\mathcal{C}$ is *mediating* iff whenever $\mathsf{red} \in \mathsf{c}_j$ and $\mathsf{blue} \in \mathsf{c}_j$ then $\mathsf{green} \in \mathsf{c}_j$.

Colour flow The coloured semantics can also be used to express the property that no output on out_1 should depend on the second wrapped component. Say $\mathcal{C}$ has the *colour directionality* property if whenever there is a reduction sequence as above and $Q \equiv (\boldsymbol{\nu}\, A)(\mathsf{c}\!:\!\overline{out_1}^{\uparrow} v \mid Q')$ then $\mathsf{red} \notin \mathsf{c}$.

For an example wrapper that we conjecture has the permutation property but not the colour directionality property, consider a version of $\mathcal{W}_2$ that has an extra parallel component $out_2{}^{a_2} y.(\overline{out_2}^{\uparrow} y \mid out_1{}^{a_1} y.\overline{out_1}^{\uparrow} y)$. This establishes an additional one-shot forwarder for out_1 after forwarding a message on out_2.

These statements of mediation and coloured directionality share a defect: the use of a reduction semantics makes it awkward to consider inputs of values containing new names that have previously been output by the wrapped components. To address this one would need a coloured labelled transition semantics, allowing e.g. a refined colour directionality property to be stated as follows. Whenever

$$A \vdash (\mathsf{gr} \circ \mathcal{C})\big[\mathsf{bl} \circ P_1,\, \mathsf{rd} \circ P_2\big] \xrightarrow{\ell_1}_{\mathsf{c}_1} \ldots \xrightarrow{\ell_k}_{\mathsf{c}_k},$$

if the inputs are properly coloured (i.e. for each $i \in 1..k$ we have $\ell_i = in_1^{\overline{\uparrow}} v \implies \mathsf{c}_i = \mathsf{blue}$ and $\ell_i = in_2^{\overline{\uparrow}} v \implies \mathsf{c}_i = \mathsf{red}$), then for each $i \in 1..k$ the out_1 outputs should be properly coloured, i.e.

$$\ell_i = \overline{out_1}^{\uparrow} v \implies \mathsf{red} \notin \mathsf{c}_i$$

Causality A very strong directionality property that one might ask for – perhaps the strongest – would be that in an execution of $\mathcal{C}[P_1, P_2]$ no output on out_1 can be *causally dependent* on any action of P_2. Casual semantics for process calculi have been much studied, often under the name 'true concurrency semantics' – see [40] for an overview. It would be interesting to give a causal semantics to the box π calculus. There is a trade-off here, however – such a semantics would be rather complex; it would have to be understood in order to understand any property stated using it. The coloured reduction semantics can be considered as an more tractable approximation to real causality.

Another point is that a causal property is sometimes too strong – a usable wrapper may have to allow low-bandwidth communication in the reverse direction, perhaps not carrying any data values, to permit acknowledgement messages. A causal property would then not hold, while a modified colour flow property would.

6 Conclusion

The code base of modern systems is becoming increasingly diverse. Whereas previously a typical system would involve a small number of monolithic applications, obtained from trusted organisations, now users routinely download components from partially trusted or untrusted sources. Downloaded or mobile code fragments are commonly run under the user's authority to grant access to system resources and permit interaction with other software components. This presents obvious security risks for the secrecy and integrity of the user's data.

In this paper we have developed a theory of security wrappers. These are small programs that can regulate the interactions between untrusted software components, enforcing dynamic and flexible security policies. We have presented a minimal concurrent programming language for studying the problem, the box-π calculus, and proved a basic metatheoretic result: that a reduction and labelled transition semantics coincide. We have expressed a number of security wrappers in the calculus and begun an investigation of the security properties that wrappers should provide.

6.1 Related Work

There is an extensive literature on information flow properties of various kinds. Much of it is in the context of multilevel security, in which one has a fixed lattice of security levels and is concerned with properties which state that a component (expressed purely semantically, e.g. as a set of traces) respects the levels. The theory could be applied during the design of the components of a large multi-user system (with a relatively static security policy) by proving that the components obey particular properties. A concise introduction can be found in the survey of McLean [23]. The problem of designing and understanding wrappers appears to be rather different – we have focussed on the protection required by a single user executing a variety of partially-trusted components obtained from third parties. This requires flexible protection mechanisms – a static assignment of security levels would be inadequate – and cannot depend on static analysis of the components. Related work on dynamic enforcement of policies has been presented by Schneider [29].

Other recent work has studied type systems that ensure security properties, e.g. the type systems of Volpano, Irvine and Smith [38, 39], the SLam calculus of Heintze and Riecke [15], the systems allowing declassification of Myers and Liskov [26, 25], the type systems of Riely and Hennessy [17, 16, 28], and work on proof-carrying code [27]. If the producers of components that one uses all adopt such systems then they may become very effective. Until then, however, and until type systems can provide the flexible policies required, partially trusted code will in practice either be run dangerously or be wrapped.

In this paper we have made extensive use of techniques from process calculi and operational semantics. These are beginning to provide fruitful ways of studying problems in security and distributed systems, including the analysis of security protocols, for example in [3, 1, 22], and more general secure language design, including work on the Ambient calculus [9, 10], the Secure Join calculus [2], the mobile agent calculi in [17, 16, 28, 30, 31, 33, 34], and the Seal calculus of [36, 37]. These works have studied several different problems, using a variety of calculi designed for the purpose. Common to all is the use of a reduction or labelled-transition operational semantics, providing clear rigorous semantics to the rather high-level constructs involved. One distinguishing feature of the present work is that we do not consider any mobility primitives, allowing us to use a tractable early labelled transition system. This appears to be important for the statement of the delicate security properties of wrappers.

6.2 Future Directions

This paper opens up a number of directions that we would like to pursue. Most immediately, it gives several conjectures that should be proved or refuted, and we would like a better understanding of the properties of binary wrappers. There are then extensions for typing, to richer interfaces, and with mobility primitives.

Typing We are primarily interested in components for which it is infeasible to statically determine whether they are well-behaved. Nonetheless, for simple components one could conservatively ensure well-behaviour with a standard type system, most simply taking types

$$T \quad ::= \quad \mathbf{box} \;\big|\; \langle T_1 .. T_k \rangle \;\big|\; \updownarrow T$$

where $\updownarrow T$ is the type of channel names that can be used to communicate values of type T, together with the obvious inference rules. If P is well-typed with respect to a typing context $in : \updownarrow S,\ out : \updownarrow T$ for types S and T containing no instances of $\updownarrow$ then one would expect P to be well-behaved for unary wrappers.

Richer interfaces The wrappers of §3 allowed the encapsulated components to interact only on very simple interfaces. Ultimately, we would like to understand wrappers with more realistic interfaces. For example, in a mild extension of box-π one can express a wrapper that encapsulates k components, allows internal flow along an arbitrary preorder, and permits each component to open and close windows for character IO. Suppose $p_1, \ldots, p_k$ is a list of distinct names, and $\geq$ is a preorder over them giving the allowable information flow. Define a k-ary wrapper as follows.

$$\begin{array}{l}
\mathcal{C}[__1, \ldots, __k] \stackrel{def}{=} \\
\quad (\boldsymbol{\nu}\, p_1, \ldots, p_k)\big(\\
\quad\quad p_1[__1] \mid \ldots \mid p_k[__k] \\
\quad\quad \mid \mathbf{!}\, fwd^{(m)}(n\, z\, y).\mathbf{if}\ m \geq n\ \mathbf{then}\ \overline{z}^n y\ \text{else}\ 0 \\
\quad\quad \mid \text{BWINDOW}\big)
\end{array}$$

where

$$
\begin{aligned}
\text{BWINDOW} \overset{\mathit{def}}{=} \; & !\, \mathit{openwindow}^{(m)}(s\,x). \\
& \quad \overline{\mathit{openwindow}}^{\uparrow}\langle s\,x\rangle \\
& \quad \mid x^{\uparrow}(\mathit{getc}\;\mathit{putc}\;\mathit{close}). \\
& \qquad \overline{x}^{m}\langle \mathit{getc}\;\mathit{putc}\;\mathit{close}\rangle \\
& \qquad \mid !\, \mathit{getc}^{m} y.(\overline{\mathit{getc}}^{\uparrow} y \mid y^{\uparrow} c.\overline{y}^{m} c) \\
& \qquad \mid !\, \mathit{putc}^{m}(c\,y).(\overline{\mathit{putc}}^{\uparrow}\langle c\,y\rangle \mid y^{\uparrow}.\overline{y}^{m}) \\
& \qquad \mid !\, \mathit{close}^{m} y.(\overline{\mathit{close}}^{\uparrow} y \mid y^{\uparrow}.\overline{y}^{m})
\end{aligned}
$$

This uses an additional input tag – a process $x^{(n)}p.P$ will input from any child box, binding the name of the box to n in P. The BWINDOW part of $\mathcal{C}$ receives requests for a new window from the encapsulated components and forwards them to the OS. It then receives the interface for the new window from the OS, forwarding it down to the component and also setting up forwarders for the interface channels. Making the security properties of $\mathcal{C}$ precise is at present a challenging problem. One would like to extend $\mathcal{C}$ further by adding an interface allowing the user to dynamically add and remove pairs from $\geq$.

Covert channels It should be noted that none of the semantic models that we use for the box-π calculus make any commitment to the precise details of scheduling processes. The properties expressed using these semantics therefore cannot address timing-based covert channels such as those mentioned by Lampson [21]. Certain other covert channels, in particular those involving system IO and disc access, could be addressed by expressing models of the IO and disc systems in the calculus, further enriching the wrapper interfaces.

Mobility The original motivation for this work involved downloadable or mobile code and mobile agents. To explicitly model the dynamic configuration of wrappers and applications the calculus must be extended with mobility primitives, while keeping both a tractable semantics and the principle that each box controls the interactions and movements of its contents [36].

Acknowledgements Sewell was supported by EPSRC grant GR/L 62290 *Calculi for Interactive Systems: Theory and Experiment*. The authors would like to thank Ciarán Bryce for his comments.

Appendix

A Coincidence of the Two Semantics

This appendix sketches the proof of equivalence of the labelled transition semantics and the reduction semantics. It is divided into three parts, the first giving basic properties of the labelled transition system, the second showing that any reduction can be matched by a τ-transition and the third showing the converse.

Basic Properties of the LTS The first lemmas are all proved by induction on derivations of transitions.

Lemma 10 *If* $A \vdash P \xrightarrow{\ell} Q$ *then*

1. $\mathrm{fn}(P) \subseteq A$ *and* $\mathrm{fn}(Q) \subseteq \mathrm{fn}(P, \ell)$.
2. *if* $\ell = \overline{x}^{o} v$ *then* $\mathrm{fn}(\ell) \cap A \subseteq \mathrm{fn}(P)$, $\mathrm{fn}(o) \subseteq \mathrm{fn}(P)$, *and moreover if* $\neg \mathrm{mv}(o)$ *then* $x \in \mathrm{fn}(P)$.
3. *if* $\ell = x^{\gamma} v$ *then* $\mathrm{fn}(\gamma) \subseteq \mathrm{fn}(P)$. *Moreover, if* $\gamma \neq \overline{n}$ *then* $x \in \mathrm{fn}(P)$.

Lemma 11 (Strengthening) *If* $A, B \vdash P \xrightarrow{\ell} P'$ *and* $B \cap \mathrm{fn}(P, \ell) = \emptyset$ *then* $A \vdash P \xrightarrow{\ell} P'$.

Lemma 12 (Injective Substitution) *If* $A \vdash P \xrightarrow{\ell} P'$, *and* $f : A \to B$ *and* $g : (\mathrm{fn}(\ell) - A) \to (\mathcal{N} - B)$ *are injective, then* $B \vdash fP \xrightarrow{(f+g)\ell} (f+g)P'$.

Lemma 13 (Shifting)

1. $(A \vdash P \xrightarrow{z^{\iota} v} P' \wedge x \in \mathrm{fn}(v) - A)$ *iff* $(A, x \vdash P \xrightarrow{z^{\iota} v} P' \wedge x \in \mathrm{fn}(v) - \mathrm{fn}(P))$.
2. $(A \vdash P \xrightarrow{z^{\overline{n}} v} P' \wedge x \in \mathrm{fn}(z, v) - A)$ *iff* $(A, x \vdash P \xrightarrow{z^{\overline{n}} v} P' \wedge x \in \mathrm{fn}(z, v) - \mathrm{fn}(P))$

As we are working up to alpha conversion some care is required when analysing transitions. We need the following lemma for transitions of an input or restricted process, together with analogous but less interesting results for the other process constructors.

Lemma 14

1. $A \vdash x^{\iota} p.P \xrightarrow{\ell} Q$ *iff there exists* v *such that* $\mathrm{fn}(x^{\iota} p.P) \subseteq A$, $\ell = x^{\iota} v$, $\{{}^{v}\!/_{p}\}P$ *is defined and* $Q \equiv \{{}^{v}\!/_{p}\}P$.
2. $A \vdash (\boldsymbol{\nu}\, x)P \xrightarrow{\ell} Q$ *iff either*

(a) *there exists* $\hat{x} \notin A \cup \mathrm{fn}(\ell) \cup (\mathrm{fn}(P) - x)$ *and* $\hat{Q}$ *such that* $A, \hat{x} \vdash \{^{\hat{x}}/_x\}P \xrightarrow{\ell} \hat{Q}$ *and* $Q \equiv (\boldsymbol{\nu}\, \hat{x})\hat{Q}$.

(b) *there exists* y, o, v, $\hat{Q}$ *and* $\hat{x} \notin A \cup \mathrm{fn}(y, o) \cup (\mathrm{fn}(P) - x)$ *such that* $\ell = \overline{y}^o v$, $A, \hat{x} \vdash \{^{\hat{x}}/_x\}P \xrightarrow{\overline{y}^o v} \hat{Q}$, $\hat{x} \in \mathrm{fn}(v)$, $\neg\mathrm{mv}(o)$ *and* $Q \equiv \hat{Q}$.

(c) *there exists* y, o, v, $\hat{Q}$ *and* $\hat{x} \notin A \cup \mathrm{fn}(o) \cup (\mathrm{fn}(P) - x)$ *such that* $\ell = \overline{y}^o v$, $A, \hat{x} \vdash \{^{\hat{x}}/_x\}P \xrightarrow{\overline{y}^o v} \hat{Q}$, $\hat{x} \in \mathrm{fn}(y, v)$, $\mathrm{mv}(o)$ *and* $Q \equiv \hat{Q}$.

Reductions Imply Transitions This direction of the equivalence has two main parts: we must show that transitions are invariant under structural congruence and construct τ-transitions for each reduction axiom.

Proposition 15 *If* $P' \equiv P$ *then* $A \vdash P' \xrightarrow{\ell} Q$ *iff* $A \vdash P \xrightarrow{\ell} Q$.

Proof A lengthy induction on the size of derivation of $P' \equiv P$. □

Lemma 16 *If* $\mathrm{fn}(P) \subseteq A$ *and* $P \to Q$ *then* $A \vdash P \xrightarrow{\tau} Q$.

Proof Induction on derivations of $P \to Q$, constructing derivations of τ-transitions for the reduction axioms (Red Up), (Red Down), (Red Comm) and (Red Repl), and using Proposition 15 for the (Red Struct) case. □

Transitions Imply Reductions For the converse direction we first show that if a process has an output or input transition then it contains a corresponding output, input or box subterm.

Lemma 17 *If* $A \vdash P \xrightarrow{\overline{z}^o v} P'$ *then* $P \equiv (\boldsymbol{\nu}\, \mathrm{fn}(z, v) - A)(\overline{z}^o v \mid P')$

Lemma 18 *If* $A \vdash Q \xrightarrow{x^\iota v} Q'$ *then there exist* B, p, Q_1 *and* Q_2 *such that* $B \cap (A \cup \mathrm{fn}(x^\iota v)) = \{\}$ *and either* $Q \equiv (\boldsymbol{\nu}\, B)(x^\iota p.Q_1 \mid Q_2)$ *and* $Q' \equiv (\boldsymbol{\nu}\, B)(\{^v/_p\}Q_1 \mid Q_2)$ *or* $Q \equiv (\boldsymbol{\nu}\, B)(!\, x^\iota p.Q_1 \mid Q_2)$ *and* $Q' \equiv (\boldsymbol{\nu}\, B)(\{^v/_p\}Q_1 \mid !\, x^\iota p.Q_1 \mid Q_2)$.

Lemma 19 *If* $A \vdash Q \xrightarrow{x^{\overline{n}} v} Q'$ *then there exist* B, Q_1 *and* Q_2 *such that* $B \cap (A \cup \mathrm{fn}(x^{\overline{n}} v)) = \{\}$, $Q \equiv (\boldsymbol{\nu}\, B)(n[Q_1] \mid Q_2)$ *and* $Q' \equiv (\boldsymbol{\nu}\, B)(n[(\overline{x}^{\uparrow} v \mid Q_1)] \mid Q_2)$.

Lemma 20 *If* $A \vdash P \xrightarrow{\tau} Q$ *then* $P \to Q$.

Proof Induction on derivations of $A \vdash P \xrightarrow{\tau} Q$, using the preceding three lemmas for the (Trans Box-1) and (Trans Comm) rules. □

The proof of Theorem 1, i.e. that if $\mathrm{fn}(P) \subseteq A$ then $A \vdash P \xrightarrow{\tau} Q$ iff $P \to Q$, is now immediate from Lemmas 16 and 20.

B Properties of $\mathcal{W}_1$

Explicit Characterisation The simple security properties of $\mathcal{W}_1$ are proved using an explicit characterisation of the states and labelled transitions of $\mathcal{W}_1[P]$. If N is a finite set of names, a is a name and $\mathcal{A}$ and Q are processes define

$$
\begin{array}{lcll}
[\![a; N; \mathcal{A}; Q]\!] & \stackrel{def}{=} & (\boldsymbol{\nu}\, N \cup \{a\})(& \mathcal{A} \\
 & & & \mid a[Q] \\
 & & & \mid !\, in^{\uparrow} y.\overline{in}^a y \\
 & & & \mid !\, out^a y.\overline{out}^{\uparrow} y\ )
\end{array}
$$

Say the 4-tuple $a, N, \mathcal{A}, Q$ is *good* if N, $\{a\}$, and $\{in, out\}$ are pairwise disjoint, $\mathcal{A}$ is a parallel composition of outputs of the forms

$$\overline{out}^{\overline{a}} v,\ \overline{out}^{\uparrow} v,\ \overline{in}^a v,\ \overline{x}^{\overline{a}} v \text{ where } x \notin \{out, a\}$$

with $a \notin \mathrm{fn}(v)$ in each case, and Q is a process with $a \notin \mathrm{fn}(Q)$. Say a process P is good if $P \equiv [\![a; N; \mathcal{A}; Q]\!]$ for some good a, N, $\mathcal{A}$, Q.

Lemma 21 *If* $a \notin \mathrm{fn}(P)$ *then* $\mathcal{W}_1[P] \equiv [\![a; \emptyset; 0; P]\!]$, *hence* $\mathcal{W}_1[P]$ *is good.*

We define a transition relation $A \vdash P \xrightharpoonup{\ell} Q$ as the least satisfying the rules in Figure 2.

Lemma 22 *For all good* P *we have* $A \vdash P \xrightarrow{\ell} P'$ *iff* $A \vdash P \xrightharpoonup{\ell} P'$. *Moreover, if* $A \vdash P \xrightharpoonup{\ell} P'$ *then* P' *is good.*

Purity Proposition 2 can now be proved by induction on k using Lemma 22.

Honesty Proposition 5, that the unary wrapper $\mathcal{W}_1$ is honest, can be proved by giving an explicit h-bisimulation. Define

$$
\begin{array}{lcl}
\langle\!\langle a; N; \mathcal{A}; Q \rangle\!\rangle & \stackrel{def}{=} & (\boldsymbol{\nu}\, N)(Q \\
 & & \mid \{\!|\ \overline{out}^{\uparrow} v \mid \overline{out}^{\overline{a}} v \in \mathcal{A}\ |\!\} \\
 & & \mid \{\!|\ \overline{out}^{\uparrow} v \mid \overline{out}^{\uparrow} v \in \mathcal{A}\ |\!\} \\
 & & \mid \{\!|\ \overline{x}^{\uparrow} v \mid \overline{x}^{\overline{a}} v \in \mathcal{A} \wedge x \neq out\ |\!\} \\
 & & \mid \{\!|\ \overline{in}^{\overline{\uparrow}} v \mid \overline{in}^a v \in \mathcal{A}\ |\!\}\ )
\end{array}
$$

$$
\begin{array}{rrcll}
 & A \vdash [\![a;N;\mathcal{A};Q]\!] & \overset{in^{\uparrow}v}{\rightharpoonup} & [\![a;N;\mathcal{A} \mid \overline{in}^{a}v;Q]\!] & \mathrm{fn}(v) \cap (N \cup \{a\}) = \emptyset \\
 & A \vdash [\![a;N;\mathcal{A} \mid \overline{in}^{a}v;Q]\!] & \overset{\tau}{\rightharpoonup} & [\![a;N;\mathcal{A};Q \mid \overline{in}^{\uparrow}v]\!] & \\
A,N,a \vdash Q \xrightarrow{\overline{out}^{\uparrow}v} Q' & A \vdash [\![a;N;\mathcal{A};Q]\!] & \overset{\tau}{\rightharpoonup} & [\![a;N,\mathrm{fn}(v)-(A,N,a);\mathcal{A} \mid \overline{out}^{\overline{a}}v;Q']\!] & \\
A,N,a \vdash Q \xrightarrow{\overline{x}^{\uparrow}v} Q' & A \vdash [\![a;N;\mathcal{A};Q]\!] & \overset{\tau}{\rightharpoonup} & [\![a;N,\mathrm{fn}(x,v)-(A,N,a);\mathcal{A} \mid \overline{x}^{\overline{a}}v;Q']\!] & \\
 & A \vdash [\![a;N;\mathcal{A} \mid \overline{out}^{\overline{a}}v;Q]\!] & \overset{\tau}{\rightharpoonup} & [\![a;N;\mathcal{A} \mid \overline{out}^{\uparrow}v;Q]\!] & \\
 & A \vdash [\![a;N;\mathcal{A} \mid \overline{out}^{\uparrow}v;Q]\!] & \overset{\overline{out}^{\uparrow}v}{\rightharpoonup} & [\![a;N-\mathrm{fn}(v);\mathcal{A};Q]\!] & \\
A,N,a \vdash Q \xrightarrow{\tau} Q' & A \vdash [\![a;N;\mathcal{A};Q]\!] & \overset{\tau}{\rightharpoonup} & [\![a;N;\mathcal{A};Q']\!] & \\
A \vdash P \overset{\ell}{\rightharpoonup} P' \equiv P'' & A \vdash P & \overset{\ell}{\rightharpoonup} & P'' &
\end{array}
$$

For all rules we have a sidecondition that the 4-tuple in the left hand side of the conclusion is good and that the free names of the process on the left hand side of the conclusion are contained in A. For the fourth rule we have a side condition that $x \neq out$.

Figure 2. Explicit Characterisation of the Transitions of $\mathcal{W}_1[P]$

Now take the family of relations below.

$$
\begin{aligned}
R_A = \equiv \circ \{ [\![a;N;\mathcal{A};Q]\!], \langle\!\langle a;N;\mathcal{A};Q \rangle\!\rangle \quad | \\
a;N;\mathcal{A};Q \text{ good and } \mathrm{fn}([\![a;N;\mathcal{A};Q]\!]) \subseteq A\} \circ \equiv
\end{aligned}
$$

One can check that for any P with $a \notin \mathrm{fn}(P)$ and $A \supseteq \mathrm{fn}(\mathcal{W}_1[P])$ we have $\mathcal{W}_1[P] \;\; R_A \;\; P$ and that R is an h-bisimulation.

References

[1] M. Abadi. Secrecy by typing in security protocols. In *TACS '97 (open lecture), LNCS 1281*, pages 611–638, Sept. 1997.

[2] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. In *LICS 98 (Indiana)*, pages 105–116. IEEE, Computer Society Press, July 1998.

[3] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security, Zürich*, pages 36–47. ACM Press, Apr. 1997.

[4] R. M. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proc. COORDINATION 97, LNCS 1282*, 1997.

[5] R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. In U. Montanari and V. Sassone, editors, *CONCUR '96*, volume 1119 of *Lecture Notes in Computer Science*, pages 147–162. Springer-Verlag, 1996.

[6] R. M. Amadio and S. Prasad. Localities and failures. In P. S. Thiagarajan, editor, *Proceedings of* 14^{th} *FST and TCS Conference, FST-TCS'94. LNCS 880*, pages 205–216. Springer-Verlag, 1994.

[7] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Java operating systems: Design and implementation. Technical Report UUCS-98-015, University of Utah, Department of Computer Science, Aug. 6, 1998.

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

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

[10] L. Cardelli and A. D. Gordon. Types for mobile ambients. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, 1999.

[11] B. Ford, M. Hibler, J. Lepreau, P. Tullman, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 137–151, Berkeley, CA, USA, Oct. 1996. USENIX.

[12] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proceedings of CONCUR '96. LNCS 1119*, pages 406–421. Springer-Verlag, Aug. 1996.

[13] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Sixth USENIX Security Symposium*, San Jose, California, July 1996.

[14] L. Gong. Java security architecture (JDK 1.2). Technical report, JavaSoft, July 1997. Revision 0.5.

[15] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th POPL*, Jan. 1998.

[16] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In *Workshop on High-Level Concurrent Languages*, 1998. Full version as University of Sussex technical report CSTR 98/02.

[17] M. Hennessy and J. Riely. Type-safe execution of mobile agents in anonymous networks. In *Workshop on Mobile Object Systems, (satellite of ECOOP '98)*, 1998. Full version as University of Sussex technical report CSTR 98/03.

[18] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proceedings of ECOOP '91, LNCS 512*, pages 133–147, July 1991.

[19] N. Islam, R. Anand, T. Jaeger, and J. R. Rao. A flexible security system for using Internet content. *IEEE Software*, 14(5):52–59, Sept./Oct. 1997.

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

[21] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[22] G. Lowe and B. Roscoe. Using CSP to detect Errors in the TMN Protocol. *IEEE Transactions on Software Engineering*, 23(10):659–669, 1997.

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

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

[25] A. C. Myers. Jflow: Practical static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL 99)*, 1999.

[26] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy, Oakland, California*, pages 186–197, 1998.

[27] 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.

[28] J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *Proceedings of the 25th POPL*, Jan. 1998.

[29] F. B. Schneider. Enforceable security policies. Technical Report TR 98-1664, Computer Science Department, Cornell University, Ithaca, New York, Jan. 1998.

[30] P. Sewell. Global/local subtyping for a distributed π-calculus. Technical Report 435, University of Cambridge, Aug. 1997. Available from `http://www.cl.cam.ac.uk/users/pes20/`.

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

[32] P. Sewell. A brief introduction to applied π, Jan. 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/`.

[33] P. Sewell, P. T. Wojciechowski, and B. C. Pierce. Location independence for mobile agents. In *Workshop on Internet Programming Languages, Chicago*, May 1998.

[34] P. Sewell, P. T. Wojciechowski, and B. C. Pierce. Location-independent communication for mobile agents: a two-level architecture. Submitted for publication. Draft available from `http://www.cl.cam.ac.uk/users/pes20/`, 1998.

[35] J. Vitek and C. Bryce. Secure mobile code: the javaseal experiment. Manuscript, 1999.

[36] J. Vitek and G. Castagna. Towards a calculus of mobile computations. In *Workshop on Internet Programming Languages, Chicago*, May 1998.

[37] J. Vitek and G. Castagna. Mobile Agents and Hostile Hosts. In *Journées Francophones des Langaages Applicatifs (JFLA99)*, Morizine, France, Feb 1999.

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

[39] D. Volpano and G. Smith. Confinement properties for programming languages. *SIGACT News*, 29(3):33–42, Sept. 1998.

[40] 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.