A Fully Abstract Encoding of the π-Calculus with Data Terms[⋆]

(Extended Abstract)

Michael Baldamus[1,⋆⋆], Joachim Parrow[2], and Björn Victor[2]

[1] Linnaeus Centre for Bioinformatics
[2] Department of Information Technology, Uppsala University, Sweden

Abstract. The π-calculus with data terms (πT) extends the pure π-calculus by data constructors and destructors and allows data to be transmitted between agents. It has long been known how to encode such data types in π, but until now it has been open how to make the encoding *fully abstract*, meaning that two encodings (in π) are semantically equivalent precisely when the original πT agents are semantically equivalent. We present a new type of encoding and prove it to be fully abstract with respect to may-testing equivalence. To our knowledge this is the first result of its kind, for any calculus enriched with data terms. It has particular importance when representing security properties since attackers can be regarded as may-test observers. Full abstraction proves that it does not matter whether such observers are formulated in π or πT, both are equally expressive in this respect. The technical new idea consists of achieving full abstraction by encoding data as table entries rather than active processes, and using a firewalled central integrity manager to ensure data security.

1 Introduction

The increasingly complicated mechanisms to guarantee secure communications have spurred the development of appropriate formal description techniques. In this paper we study a prototypical such formalism, a π-calculus enriched with data terms (πT), and show how it can be encoded in the more fundamental π-calculus [13] while preserving full abstraction. This means that two processes in πT are equivalent precisely when their encodings are equivalent. Although encodings between such calculi has proved a rich field of study this is the first result of its kind. We achieve it by designing the encoding in a different way from what is usually done. The full proof is very technical and we here outline the main ideas, which are quite general and can in principle be applied to many similar calculi.

πT extends the basic π-calculus by including constructors and deconstructors for data terms. In this the calculus resembles a high level parallel programming language, and specifications of security protocols can be made in a familiar operational style. It

⋆ Work supported by European Union project PROFUNDIS, Contract No. IST-2001-33100.

⋆⋆ To whom correspondence should be addressed (*http://www.lcb.uu.se/∼michaelb*).

can be seen as a generalisation of the spi-calculus [3] (which extends π with primitives for encryption and decryption) in that arbitrary constructors and destructors are allowed. It can also be seen as a simplification of the applied π-calculus by Abadi and Fournet [1], since it does not admit equations over the data terms.

A key idea when encoding data was expressed already in the first papers on the π-calculus: a data structure is represented in π as a process with the ability to interact along designated ports in order to carry out operations. In such a manner all known kinds of data structures, certainly including those of $\pi\mathrm{T}$, can be encoded.

The problem with this kind of encoding appears when we consider a useful notion of semantic equality between $\pi\mathrm{T}$ processes. We shall here as a prime example look at *testing* equivalence. The main idea is that two processes are behaviourally equivalent precisely when they can satisfy the same *tests*. Tests are arbitrary processes of the calculus, which of course is expressive enough to describe not only security protocols but also the potential attackers on them. So if we know of two processes that they are testing equivalent we know that they will behave similarly in all conceivable environments. In an earlier paper [4] we have proved for the spi calculus that a process satisfies a test if and only if the encoding of the process satisfies the encoding of the test. The same result, as we will show in this paper, holds for an encoding from $\pi\mathrm{T}$ to π.

Unfortunately, with the traditional kind of encoding from $\pi\mathrm{T}$ to π the testing equivalences in $\pi\mathrm{T}$ and π turn out to be different. The reason is, briefly put, that the encodings of two $\pi\mathrm{T}$ processes satisfy the same tests *only if the tests are encodings of* $\pi\mathrm{T}$ *tests*. But in the π-calculus there are also tests which are not encodings of any $\pi\mathrm{T}$ test, and some of these may be able to discriminate between the encodings of otherwise equivalent processes. Formally *full abstraction*, the result that two $\pi\mathrm{T}$ processes are equivalent precisely when their encodings are equivalent, has proved elusive, not only for $\pi\mathrm{T}$ but for all similar calculi, and for similar operationally defined equivalences.

In this paper we exhibit an encoding from $\pi\mathrm{T}$ to π and prove it to be fully abstract. A key ingredient is that we use a central so called *integrity manager* ($\mathbf{M}$) which stores all data values generated throughout a computation. All access to data must go through a level of indirection at $\mathbf{M}$, which only allows accesses that adheres to the protocols for interacting with data. Thus, the previously dangerous π-calculus processes that are not encodings of $\pi\mathrm{T}$ processes are rendered impotent.

The remaining sections of this paper are organised as follows: Section 2 introduces $\pi\mathrm{T}$. Section 3 presents the encoding of $\pi\mathrm{T}$ into the polyadic π-calculus. The reader may refer to [5] for some parts that have to be left out here due to a lack of space. Section 4 states the full-abstraction result. It also gives an idea of its long and complex proof. The full proof can be found in [5]. Section 5 discusses related work. Section 6 concludes the paper with some final remarks, in particular about the lack of compositionality of the encoding due to the global integrity manager.

2 Background: π-Calculus, $\pi\mathrm{T}$-Calculus

2.1 The $\pi\mathrm{T}$-Calculus

As always in π-like calculi, an infinite set of *names* is assumed to be given. This set is here typically ranged over by lower-case letters from the middle of the alphabet, such as n. Sets of names are typically ranged over by upper-case letters from the middle of the alphabet, such as N. As usual in spi-calculus-like extensions of the π-calculus we distinguish names and *variables*. The set of variables is also assumed to be infinite. It is here typically ranged over by lower-case letters from the end of the alphabet, such as x. We designate finite vectors by means of the $\widetilde{\ }$-symbol. The length of any finite vector $\widetilde{X}$ is denoted by $|\widetilde{X}|$.

The πT-calculus is further characterised by assuming a set of function symbols from which data terms can be formed, distinguishing dedicated *constructors* and *deconstructors* (with minor restrictions, see below). Predictably, constructors are used to build data terms and deconstructors to take them apart. Both the set of constructors and the set of deconstructors are assumed to be finite. Constructors are typically ranged over by f, deconstructors by d. Each constructor f is assumed to have a fixed, finite *arity* $\mathrm{ar}[f]$.

Data terms are then given as follows:

$$T, \ldots ::= n \mid x \mid f(T_1, \ldots, T_{\mathrm{ar}[f]})$$

For the sake of simplicity, no type discipline is assumed for data terms, only the constructor arities must be respected. The set of all names that occur in any given vector $\widetilde{T}$ of data terms is denoted by $\mathrm{nm}[\widetilde{T}]$. Deconstructors must not occur within data terms. They may only be applied via the **let** construct introduced below. A *value* is a data term without any variables. Values are typically ranged over by V and W.

Agent expressions are then given as follows:

$$\begin{aligned} P, \ldots ::= &\ \mathbf{0} \mid T(x).P \mid \overline{T}\langle U\rangle.P \mid P \mid Q \mid (\nu\, n)\, P \mid !\, P \\ &\mid \mathbf{if}\, T = U\, \mathbf{then}\, P\, \mathbf{else}\, Q \mid \mathbf{let}\, x = d(\widetilde{T})\, \mathbf{in}\, P \end{aligned}$$

The **0** constant denotes the inert process. An input prefix $T(x).P$ can be performed in a context where T evaluates to some name n. Then a value, e.g. V, is received over the channel denoted by n and the process continues as P where x is substituted by V. An output prefix $\overline{T}\langle U\rangle.P$ can also be performed in a context where T evaluates to some name n and U to some value V. Then V is sent via the channel denoted by n and the process continues as P. Here n and T are called the *subject*, and x and V the *object* of the input or output. The next three operators are standard in pi-calculus-like calculi: $P \mid Q$ behaves like P and Q acting concurrently; $(\nu\, n)\, P$ behaves like P where n is local; $!\, P$ behaves like infinitely many copies of P put in parallel. The conditional and the **let** have their usual meaning. In πT, **let** is the only place where deconstructors may be applied.

Input prefixing, **let** , and restriction are *binders*: $T(x).P$ and **let** $x = d(\widetilde{T})$ **in** P bind the variable x in P; $(\nu\, n)\, P$ binds the name n in P. The set of names occurring free (non-bound) in a process expression P is denoted by $\mathrm{fn}[P]$ and similarly for a vector $\widetilde{P} = P_1 \ldots P_n$ of process expressions $\mathrm{fn}[\widetilde{P}]$ means the union of all $\mathrm{fn}[P_i]$.

We do not distinguish between expressions that differ only up to alpha conversion of bound names and variables. Given any data term or process expression H, $H\{\widetilde{x} := \widetilde{T}\}$ denotes the term after simultaneously substituting each x_i by T_i. Substitution always entails implicit alpha-conversion of bound names in H such that there is no capture of any free names in $\widetilde{T}$. A *process* is an agent expression without free variables.

We often omit trailing $\mathbf{0}$ suffixes, writing $x\langle y\rangle$ for $x\langle y\rangle.\mathbf{0}$. We often also use an "inline" notation for restriction, and e.g. write $\overline{x}\langle \nu\, y\rangle.P$ as shorthand for $(\nu\, y)\,\overline{x}\langle y\rangle.P$. Further, we use the standard notation $\prod_{i\in\{j_1,\ldots,j_k\}} P_i = P_{j_1} \mid \ldots \mid P_{j_k}$.

Deconstructor Equations. *Deconstructor equations* describe how deconstructors act upon values. To this end, we need to introduce *value patterns*:

$$G ::= x \;\big|\; f(G_1,\ldots,G_{\mathrm{ar}[f]})$$

A deconstructor equation is then of the form $d(\widetilde{G}) \overset{\Delta}{=} x$ where x must occur in $\widetilde{G}$. There may be several, but only finitely many equations for each deconstructor. Since there are finitely many deconstructors, there are only finitely many deconstructor equations.

Operational Semantics. Abadi's and Fournet's semantics for the applied π-calculus [1] use active substitutions and rely heavily on structural congruence rules. Our semantics, using the **let** construct for deconstruction, is more direct and does not utilise structural congruence. The rules for scope opening are similar to the variant in [1] giving the finest bisimulation equivalence relation; we are dealing with may testing where this finesse does not matter.

Actions are of one of three forms:

$n(V)$: Input of value V on channel n where V is bound to x as shown below.
$(\nu\, M)\,\overline{n}\langle V\rangle$: Output of value V on channel n where the names in M are extruded as private names. The SOS clauses ensure that we always have $M \subseteq \mathrm{fn}(V)$.
τ: Silent action.

The individual clauses are as shown below. The treatment of replication follows [17]. The clauses for **if** _ **then** _ **else** are slightly non-standard since they entail a τ-action, just like in [1]. This is advantageous for our purposes of considering may-testing. The missing symmetric clauses for interleaving and closure are left implicit.

$$n(x).P \xrightarrow{n(V)} P\{x := V\} \qquad \frac{P \xrightarrow{n(V)} P' \quad Q \xrightarrow{(\nu\, M)\,\overline{n}\langle V\rangle} Q' \quad \mathrm{fn}[P]\cap M = \emptyset}{P \mid Q \xrightarrow{\tau} (\nu\, M)(P' \mid Q')}$$

$$\overline{n}\langle V\rangle.P \xrightarrow{(\nu\,\emptyset)\,\overline{\mathrm{n}}\langle \mathrm{V}\rangle} P$$

$$\frac{P \xrightarrow{\alpha} P' \quad \mathrm{bn}[\alpha]\cap\mathrm{fn}[Q] = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \frac{P \xrightarrow{(\nu\, M)\,\overline{n}\langle V\rangle} P' \quad n \neq m' \quad m' \in \mathrm{nm}[V]\setminus M}{(\nu\, m')\,P \xrightarrow{(\nu\, M + m')\,\overline{n}\langle V\rangle} P'}$$

$$\frac{P \xrightarrow{\alpha} P' \quad n \notin \mathrm{nm}[\alpha]}{(\nu\, n)\,P \xrightarrow{\alpha} (\nu\, n)\,P'} \qquad \mathbf{if}\; T = T\;\mathbf{then}\; P\;\mathbf{else}\; Q \xrightarrow{\tau} P \qquad \frac{P \xrightarrow{\alpha} P' \quad \mathrm{bn}[\alpha]\cap\mathrm{fn}[P] = \emptyset}{!\,P \xrightarrow{\alpha} P' \mid !\,P}$$

$$\frac{T \neq U}{\mathbf{if}\; T = U\;\mathbf{then}\; P\;\mathbf{else}\; Q \xrightarrow{\tau} Q}$$

$$\frac{P \xrightarrow{n(V)} P_1' \quad P \xrightarrow{(\nu\, M)\,\overline{n}\langle V\rangle} P_2' \quad \mathrm{fn}[P]\cap M = \emptyset}{!\,P \xrightarrow{\tau} ((\nu\, M)(P_1' \mid P_2')) \mid !\,P} \qquad \frac{d(\widetilde{G}) \overset{\Delta}{=} x \quad \widetilde{G}\sigma = \widetilde{V}}{\mathbf{let}\; y = d(\widetilde{V})\;\mathbf{in}\; P \xrightarrow{\tau} P\{y := x\sigma\}}$$

We denote by $\mathrm{nm}[\alpha]$ the set of names that syntactically occur in any action α; we denote by $\mathrm{bn}[\alpha]$ the set of bound names of α: $\mathrm{bn}[n(x)] = \emptyset$, $\mathrm{bn}[(\nu\, M)\,\overline{n}\langle T\rangle] = M$, $\mathrm{bn}[\tau] = \emptyset$; $(\nu\,\{n_1,\ldots,n_k\})$ stands for $(\nu\, n_1)\ldots(\nu\, n_k)$, $k \geq 0$. Also, we denote mappings from variables to values by σ, and by $\widetilde{T}\sigma$ the vector that results from applying σ to each element of $\widetilde{T}$.

2.2 Polyadic π-Calculus

In the next section we will encode $\pi\mathrm{T}$ into the polyadic π-calculus. The specific dialect that we use is derived from $\pi\mathrm{T}$ via three simple modifications: First, the sets of constructors and deconstructors are assumed to be empty; second, nondeterministic CCS-like choice of the form $P + Q$ and polyadic input and output prefixes of the form $a(\widetilde{x}).P$ or $\overline{a}\langle\widetilde{b}\rangle$, respectively, are admitted, where a and b range over both names and variables; third, *process constants* are admitted. They are typically ranged over by upper-case letters from the beginning of the alphabet, such as A. *Process constant definitions* are of the form $A(\widetilde{x}) \stackrel{\Delta}{=} P$ where the free variables of P must be from $\widetilde{x}$. There must be a unique definition for each process constant. The number of actual parameters in each instantiation of any process constant must be the same as the number of formal parameters in the constant's definition. The necessary modifications of the above SOS clauses and the accompanying notion of action are standard. Here, we just give the semantics of the nondeterministic choice operator:

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

As α ranges over all actions including τ, $+$ is the ordinary CCS-like nonderministic choice operator.

3 The Encoding of the $\pi\mathrm{T}$-Calculus

In this section we present the encoding of $\pi\mathrm{T}$-processes into the polyadic π-calculus. As in previous work [4], the main issue is encoding values.

The idea is to let encoded processes operate on encoded values only via *value identifiers* (IDs). To this end, an *integrity manager* $\mathbf{M}$ is set up, which maintains tables of the existing IDs and the values these correspond to. Whenever an encoded process wants to operate on one or more encoded values, it must send a specific request carrying the value IDs to $\mathbf{M}$. $\mathbf{M}$ will check whether it already knows the IDs, i.e., whether the IDs correspond to any actual values: if that is the case, then $\mathbf{M}$ will perform the operation; if not, then the request will deadlock. If the operation requires generating a new value, $\mathbf{M}$ will do so, and it will also generate a new ID that it will associate with the value. Completion will be signalled on a dedicated channel that has been supplied when the request was issued. The channel carries an ID which refers to the result of the operation, if there was any.

$\mathbf{M}$ thus shields encoded processes from any malformed complex values that might be sent to them by π-observers which are not encodings of $\pi\mathrm{T}$ processes. Moreover, as $\pi\mathrm{T}$- and π-processes have the same pure synchronisation capabilities, π-observers are unable to do anything to encoded $\pi\mathrm{T}$-processes that encoded $\pi\mathrm{T}$-observers cannot already do. The only requirement is that no encoded $\pi\mathrm{T}$-process may ever get connected to a "fake" integrity manager. This is assured by (a) restricting the channels that are used for interacting with $\mathbf{M}$ and (b) setting up a uni-directional firewall $\mathbf{F}$ via which $\mathbf{M}$ can receive requests from an external observer via duplicate, non-restricted channels.

The encoding of $\pi\mathrm{T}$ processes is constructed using one encoding function $(\!|\cdot|\!)$ for processes as such (section 3.1), and one encoding function $(\!|\cdot|\!)_r$ of values occurring in the processes (section 3.2), parameterised by a location r used to represent the value. $\mathbf{M}$ is presented in section 3.3.

The full encoding of a πT process P is thus $[\![P]\!] = (\nu\,\mathrm{K}_{\mathbf{M}})((\!|P|\!) \mid \mathbf{M} \mid \mathbf{F})$, and our main theorem

$$P \eqsim_{\mathrm{may}} Q \text{ iff } [\![P]\!] \eqsim_{\mathrm{may}} [\![Q]\!].$$

The encoded processes communicate with $\mathbf{M}$ using a simple protocol over the free names of $\mathbf{M}$, which we denote by $\mathrm{K}_{\mathbf{M}}$. Below we describe their use and the parameters passed over them.

$\mathit{input} : (i, r)$ i is the ID of the channel the process wants to input a value over; r is a fresh name over which the process will receive the value (itself an ID).

$\mathit{output} : (i, j, r)$ i is the ID of the channel the process wants to output a value over; j is the ID of the value to be sent; r is a fresh name which will be used for synchronisation when the output has been performed by $\mathbf{M}$.

$\mathit{apply}_E : (\widetilde{i}, r)$ (for each deconstructor equation $E : d(G_1, \ldots, G_k) = x$, where $|\widetilde{i}| = k$). $\widetilde{i}$ are the IDs of the actual components of the value pattern; r is a fresh name where the ID of the value corresponding to x (if any) can be received by the process.

$r : (i)$ (for each value occurring in the process). Each value used in the process is represented by a fresh name r, where the process can receive the ID of the value, e.g. to pass in the above operations.

The encoded values register with $\mathbf{M}$ by communicating with it over two additional channels:

$\mathit{new} : (n, r)$ n is a πT name occurring in the source process; r is a name where the ID of the name can be received.

$\mathit{new}_f : (\widetilde{x}, r)$ (for each constructor f, where $|\widetilde{x}| = \mathrm{ar}[f]$). $\widetilde{x}$ are the actual components of the constructor f; r is a name where the ID of the constructed value $f(\widetilde{x})$ can be received.

3.1 Encoding Processes

The encoding of processes is homomorphic for $\mathbf{0}$, $|$, (ν) and $!$ operators:

$$(\!|\mathbf{0}|\!) = \mathbf{0} \quad (\!|P \mid Q|\!) = (\!|P|\!) \mid (\!|Q|\!) \quad (\!|(\nu\, n)\, P|\!) = (\nu\, n)(\!|P|\!) \quad (\!|!\, P|\!) = !(\!|P|\!)$$

Input and output prefixes (below) encode the subject, request its ID, and contacts $\mathbf{M}$ for doing the actual input or output. In the case of input, the response channel s is used for performing the input, binding the object x, while in the case of output, $\mathbf{M}$ performs the output and responds with a pure synchronisation. r, s, t, i and j are fresh.

$$\begin{aligned}(\!|T(x).P|\!) &= (\nu\, r)((\!|T|\!)_r \mid r(i).\overline{\mathit{input}}\langle i, \nu\, s\rangle.s(x).(\!|P|\!))\\ (\!|\overline{T}\langle U\rangle.P|\!) &= (\nu\, r, s)((\!|T|\!)_r \mid (\!|U|\!)_s \mid r(i).s(j).\overline{\mathit{output}}\langle i, j, \nu\, t\rangle.t().(\!|P|\!))\end{aligned}$$

The conditional retrieves the IDs of the encodings of the compared values and tests them for identity. ($\mathbf{M}$ ensures that identical values have the same ID.) r, s, i and j are fresh.

$$(\!|\mathbf{if}\, T = U\, \mathbf{then}\, P\, \mathbf{else}\, Q|\!) = (\nu\, r, s)((\!|T|\!)_r \mid (\!|U|\!)_s \mid r(i).s(j).\mathbf{if}\, i = j\, \mathbf{then}\, (\!|P|\!)\, \mathbf{else}\, (\!|Q|\!))$$

$$
\begin{aligned}
\mathbf{M} = {} & (\nu\, \mathit{lock}, \mathit{unlock}, \mathit{put}, \mathit{getId}, \mathit{getAlias}, [\forall f \in \mathbb{F}.\, \mathit{put}_f, \mathit{getId}_f, \mathit{getArgIds}_f], \mathit{lookUp})(\\
& \mathbf{EmptyValTbl}(\mathit{put}, \mathit{getId}, \mathit{getAlias}, [\forall f \in \mathbb{F}.\mathit{getId}_f, \mathit{getArgIds}_f], \mathit{lookUp}) \\
& \mid \mathbf{NameRegistrar}(\mathit{lock}, \mathit{unlock}, \mathit{put}, \mathit{getId}, \mathit{lookUp}, \mathit{new}) \\
& \mid \textstyle\prod_{f\in\mathbb{F}} \mathbf{ConsTermRegistrar}_f(\mathit{lock}, \mathit{unlock}, \mathit{put}_f, \mathit{getId}_f, \mathit{new}_f) \\
& \mid \mathbf{InputInterpreter}(\mathit{lock}, \mathit{unlock}, \mathit{getAlias}, \mathit{input}) \\
& \mid \mathbf{OutputInterpreter}(\mathit{lock}, \mathit{unlock}, \mathit{getAlias}, \mathit{lookUp}, \mathit{output}) \\
& \mid \textstyle\prod_{E\in\mathbb{E}} \mathbf{EquationInterpreter}_E(\mathit{lock}, \mathit{unlock}, [\forall f \in \mathbb{F}.\mathit{getArgIds}_f], \mathit{apply}_E) \\
& \mid \mathbf{Mutex}(\mathit{lock}, \mathit{unlock}))
\end{aligned}
$$

Fig. 1. Defining equation of integrity manager and empty value table

The **let** processes use a deconstructor. Each deconstructor d of arity k has a set of associated deconstructor equations $\mathbb{E}_{d,k}$. Each such equation E has a handler in $\mathbf{M}$, contacted over apply_E. In the encoding of **let** below (left), all such handlers are applied in parallel (line 2), and the first one to complete (line 3) locks the others out (using the one-time lock u on line 1). $\widetilde{r}, \widetilde{i}$ and u are fresh.

$$
\begin{aligned}
& (\!|\mathbf{let}\ x = d(T_1, \ldots, T_k)\ \mathbf{in}\ P|\!) = \\
& \quad (\nu\, r_1, \ldots, r_k)(\\
& \quad\quad (\!|T_1|\!)_{r_1} \mid \ldots \mid (\!|T_k|\!)_{r_k} \\
& \quad \mid\ r_1(i_1). \ \ldots\ r_k(i_k). \\
& \quad\quad (\nu\, u)(\\
& \quad\quad\quad \overline{u}\langle\rangle && (1) \\
& \quad\quad \mid\ \textstyle\prod_{E\in\mathbb{E}_{d,k}}(\\
& \quad\quad\quad \overline{\mathit{apply}_E}\langle i_1, \ldots, i_k, \nu\, r\rangle. && (2) \\
& \quad\quad\quad r(x).u().(\!|P|\!)) && (3)
\end{aligned}
$$

$$
\begin{aligned}
& (\!|f(T_1, \ldots, T_{\mathrm{ar}[f]})|\!)_r \\
& \quad = (\nu\, r_1, \ldots, r_{\mathrm{ar}[f]})(\\
& \quad\quad (\!|T_1|\!)_{r_1} \mid \ldots \mid (\!|T_{\mathrm{ar}[f]}|\!)_{r_{\mathrm{ar}[f]}} && (4) \\
& \quad \mid\ r_1(i_1). \ \ldots\ r_{\mathrm{ar}[f]}(i_{\mathrm{ar}[f]}). && (5) \\
& \quad\quad \overline{\mathit{new}_f}\langle i_1, \ldots, i_{\mathrm{ar}[f]}, r\rangle) && (6) \\
& (\!|n|\!)_r = \overline{\mathit{new}}\langle n, r\rangle \\
& (\!|x|\!)_r = \overline{r}\langle x\rangle
\end{aligned}
$$

3.2 Encoding Value Terms

Value terms come in three kinds: constructor terms, names, and variables. Their encoding is above (right). Terms using a constructor f encode their component values (line 4), and supply their IDs (line 5) to the new_f handler (line 6). Names are encoded by calling the new handler of $\mathbf{M}$, while occurrences of variables will always be substituted at runtime by an ID, which will then be returned.

3.3 Integrity Manager M

$\mathbf{M}$ is shown in Figure 1. The names input, output, new, new_f for all $f \in \mathbb{F}$, and apply_E for all $E \in \mathbb{E}$ are free in $\mathbf{M}$. They may be used by π-calculus observers to interact with any encoded process, but thanks to the firewall, they can not interfere in the communication between the encoded processes and $\mathbf{M}$.

$\mathbf{M}$ uses an initially empty value table ($\mathbf{EmptyValTbl}$) to maintain the correspondence between values and their value IDs. Each name in the table has an *alias*, used for the actual input and output, such that $\mathbf{M}$ can monitor all values passed, making sure they are value IDs. For each constructor in the table, the IDs of its subcomponents are maintained. The $\mathbf{Mutex}$ is used as a mutual exclusion lock for the table. (The reader may refer to [5] for parts that have to be left out here due to a lack of space.)

Additional components of $\mathbf{M}$ insert new values into the tables, if necessary:

$\mathbf{NameRegistrar.}$ The first time a name is used in the encoded process, the name registrar adds its corresponding ID and alias to the name table. Later uses of the name only result in a lookup; the ID and alias is maintained.

$\mathbf{ConsTermRegistrar}_f$ (for each $f \in \mathbb{F}$). The constructor term registrars perform the corresponding function for values built by constructors f; the ID of the value and its components is maintained, and together the name and constructor term registrars ensure that identical values have the same ID.

The remaining components of $\mathbf{M}$ are (1) the handlers for input and output, $\mathbf{Input}$-$\mathbf{Interpreter}$ and $\mathbf{OutputInterpreter}$, which ensure that all names used as channels, and values passed over them, have appropriate table entries and (2) the deconstructor equation handlers $\mathbf{EquationInterpreter}_E$, used to match a deconstructor application against deconstructor equations.

Value Table. IDs of names and constructors are added to the value table by communication over put and put_f, respectively; they can be retrieved over *getId* and *getId*$_f$. The *lookUp* is used to verify that an ID is in the table. Subcomponents of constructors can be retrieved over $getArgIds$.

The table is built by appending table cells to the initial empty table. Such a cell is either a $\mathbf{NameEntry}$ or a $\mathbf{ConsTermEntry}_f$ ($f \in \mathbb{F}$). All requests to retrieve information have among their parameters the names r^+, r^-, which are used for the response. Each cell checks if it should handle the request, and if not, passes it on to the next cell. If no cell handles the request, the $\mathbf{EmptyValTbl}$ eventually signals on r^-.

Name Registrar. Name records maintain the IDs and *aliases* of names. The latter are the channels used internally for communication, to make sure all objects passed are value IDs. The *name registrar* first checks if the supplied name is a value ID, supplied by a "malicious" observer: this is an error, and the requester gets no reply. If it is not a value ID, it checks if the name is in the name table; in that case it returns its ID, otherwise it adds the name together with its new ID and alias to the name table, adds the ID to the ID table (since all IDs used must be there), and responds with the ID.

Constructor Term Registrars. The constructor registrars for a constructor f is a variation of the name registrar: it checks that each component value is in the ID table (and thus is a valid value in the encoding), and if they are, either returns the ID of the constructed term (if it is already there) or puts in an association between the constructed term and its (new) ID, puts the ID in the ID table, and returns it.

Equation Interpreters. Given an equation $E : d(G_1, \ldots, G_{\mathrm{ar}[E]}) \stackrel{\Delta}{=} x$, an interpreter for E is given a vector of $\mathrm{ar}[E]$ actual value IDs, and tries to match each value ID against the corresponding value pattern G_i of the equation. If it succeeds, the value ID corresponding to x of the equation is returned over the result channel r. If it fails, no result is given.

Given $E : d(G_1, \ldots, G_{\mathrm{ar}[E]}) \stackrel{\Delta}{=} x$,

$$
\begin{array}{l}
\mathbf{EquationInterpreter}_E(\mathit{lock}, \mathit{unlock}, [\forall f \in \mathbb{F}.\mathit{getArgIds}_f], \mathit{lookUp}, \mathit{apply}_E) \stackrel{\Delta}{=} \\
\quad !\ \ \mathit{apply}_E(i_1, \ldots, i_{\mathrm{ar}[E]}, r).\mathit{lock}().\overline{\mathit{lookUp}}\langle i_1, \nu\, s_1^+, \nu\, s_1^- \rangle \\
\qquad (\ \ s_1^-().\overline{\mathit{unlock}}\langle\rangle \\
\qquad +\ s_1^+(). \\
\qquad\qquad \vdots \\
\qquad\qquad \overline{\mathit{lookUp}}\langle i_{\mathrm{ar}[E]}, \nu\, s_{\mathrm{ar}[E]}^+, \nu\, s_{\mathrm{ar}[E]}^- \rangle. \\
\qquad\qquad (\ \ s_{\mathrm{ar}[f]}^-().\overline{\mathit{unlock}}\langle\rangle \\
\qquad\qquad +\ s_{\mathrm{ar}[f]}^+().(\!| G_1, i_1, \ldots, G_{\mathrm{ar}[E]}, i_{\mathrm{ar}[E]}, \emptyset |\!)) \cdots)
\end{array}
$$

where the $(\!| H_1, j_1, \ldots, H_k, j_k, S |\!)$-construct, $k \geq 0$, handles the matching of actual component values against components of the equation. Its definition is recursive.

1. If $k = 0$, then $(\!| S |\!) = \overline{\mathit{unlock}}\langle\rangle.\overline{r}\langle x \rangle$.
2. If $k \geq 1$ and $H_1 = z$ for some variable $z \notin S$,
   $(\!| H_1, j_1, \ldots, H_k, j_k, S |\!) = (\!| H_2, j_2, \ldots, H_k, j_k, S \cup \{z\} |\!)\{z := j_1\}$
3. If $k \geq 1$ and $H_1 = z$ for some variable $z \in S$,
   $(\!| H_1, j_1, \ldots, H_k, j_k, S |\!) = \textbf{if } z = j_1 \textbf{ then } (\!| H_2, j_2, \ldots, H_k, j_k, S |\!)$
   $\qquad\qquad\qquad\qquad\qquad \textbf{else } \overline{\mathit{unlock}}\langle\rangle$
4. If $k \geq 1$ and $H_1 = f(H'_1, \ldots, H'_{\mathrm{ar}[f]})$ for some constructor f and arguments H'_1, $\ldots$, $H'_{\mathrm{ar}[f]}$, then:
   $$
   \begin{array}{rl}
   (\!| H_1, j_1, \ldots, H_k, j_k, S |\!) = & \overline{\mathit{getArgIds}_f}\langle j_1, \nu\, t^+, \nu\, t^- \rangle. \\
   & (\ \ t^-().\overline{\mathit{unlock}}\langle\rangle \\
   & +\ t^+(j'_1, \ldots, j'_{\mathrm{ar}[f]}). \\
   & \quad (\!| H'_1, j'_1, \ldots, H'_{\mathrm{ar}[f]}, j'_{\mathrm{ar}[f]}, H_2, j_2, \ldots, H_k, j_k, S |\!))
   \end{array}
   $$
   where $j'_1, \ldots, j'_{\mathrm{ar}[f]}$ are fresh.

Communication Interpreters. The communication interpreters handle the input and output requests to $\mathbf{M}$. Both look up the alias of the subject channel in the name table; the input handler returns this so the encoded input prefix can perform the input and bind the object variable in the correct context; the output handler looks up the object ID and performs the output on behalf of the encoded output prefix, and synchronises on the result channel when done.

$$
\begin{array}{l}
\mathbf{InputInterpreter}(\\
\quad \mathit{lock}, \mathit{unlock}, \mathit{getAlias}, \mathit{input} \\
) \stackrel{\Delta}{=} !\ \ \mathit{input}(i, r).\mathit{lock}(). \\
\qquad \overline{\mathit{getAlias}}\langle i, \nu\, s^+, \nu\, s^- \rangle. \\
\qquad (\ \ s^-().\overline{\mathit{unlock}}\langle\rangle \\
\qquad +\ s^+(a).\overline{\mathit{unlock}}\langle\rangle.a(x).\overline{r}\langle x \rangle)
\end{array}
$$

$$
\begin{array}{l}
\mathbf{OutputInterpreter}(\\
\quad \mathit{lock}, \mathit{unlock}, \mathit{getAlias}, \mathit{lookUp}, \mathit{output} \\
) \stackrel{\Delta}{=} !\ \ \overline{\mathit{output}}\langle i, j, r \rangle.\mathit{lock}(). \\
\qquad \overline{\mathit{getAlias}}\langle i, \nu\, s^+, \nu\, s^- \rangle. \\
\qquad (\ \ s^-().\overline{\mathit{unlock}}\langle\rangle \\
\qquad +\ s^+(a).\overline{\mathit{lookUp}}\langle j, \nu\, t^+, \nu\, t^- \rangle \\
\qquad\quad (\ \ t^-().\overline{\mathit{unlock}}\langle\rangle \\
\qquad\quad +\ t^+().\overline{\mathit{unlock}}\langle\rangle.\overline{a}\langle j \rangle.\overline{r}\langle\rangle))
\end{array}
$$

3.4 Firewall

For every free name n of $\mathbf{M}$, we introduce a distinct fresh name n' via which external observers may interact with $\mathbf{M}$ and thus indirectly also with any encoded πT-process. The firewall $\mathbf{F}$ receives requests on the channels named with those fresh names and encodes them to requests on the corresponding internal channels.

4 Full Abstraction

The relevance of may-testing equivalence for analysing security protocols has been stated elsewhere (see e.g. [3]). For this reason, we keep this section relatively technical apart from stating an outline of the long and complex proof of the full abstraction result. Recall the full encoding of a πT process P is $[\![P]\!] = (\nu\, \mathrm{K}_{\mathbf{M}})((\!|P|\!) \mid \mathbf{M} \mid \mathbf{F})$.

Definition 1. *1. An* observer *is a process that may use a distinguished name* \$*. An action on channel* \$ *is a* success signal*.*

2. *A* test *is a parallel composition $P \mid O$ of a process P and an observer O.*
3. *A process P* may pass *a test $P \mid O$ if some sequence of τ-steps of $P \mid O$ has a state in which O signals success. Formally, we denote this property by P* may *O.*
4. *Any two processes P and Q are* may-testing equivalent *if P* may *O if and only if Q* may *O for every observer O – in other words, the tests P and Q may pass are the same. We denote this property by $P \eqsim_{\mathrm{may}} Q$.*

Theorem 2. *Given any πT-processes P and Q, $P \eqsim_{\mathrm{may}} Q$ iff $[\![P]\!] \eqsim_{\mathrm{may}} [\![Q]\!]$.*

Proof. (Outline) The proof has two main steps:

1. This step consists of proving that may-tests are preserved, that is to say, proving that the encodings of two πT processes satisfy the same tests if the tests are encodings of πT tests. This result is analogous to earlier work [4] but, at the same time, it is by far more difficult. The reason is that, unlike [4], the global context in the form of the integrity manager has to be taken into account. The solution consists of working with a set-theoretical abstraction of the integrity manager that allows us to consider only factions of it as we exploit the syntax-directed way in which the pure encoding $(\!|\_|\!)$ is defined. In this way we are able to re-instantiate the concept of an ancestor relation introduced in [4] to prove both a forward and a backward operational correspondence between unencoded and encoded process-observer couplings. The preservation of may tests is then an easy corollary once we also have an operational correspondence between abstract and concrete integrity management.

2. This step mainly consists of using the result from Step 1 in obtaining a π-calculus trace characterisation of πT-may-testing equivalence. This property is reminiscent of trace characterisations of may-testing equivalence in other settings (see e.g. [10]). All what is left is then to take the firewall into account to obtain the final full-abstraction result.

The full proof can be found in [5].

5 Related Work

Arbitrary equations between data terms are the most crucial feature of the applied π-calculus that is lacking from πT. Also, we require that deconstructors are only used in let-expressions. πT is therefore in between the applied π-calculus and the spi calculus in that it resembles Borgström, Briais and Nestmann's parameterised spi calculus [7]. These simplifications entail a loss of expressiveness when describing security protocols. The advantage is that we can still give a straightforward Structural Operational Semantics (SOS) for agents without having to first define a complex evaluation semantics for value terms. This simplifies our proofs to a great extent.

Full abstraction in the kind of encoding considered here is difficult to achieve. Milner's encoding of the λ-calculus [12] is not fully abstract. Sangiorgi [16] demonstrates full abstraction for an encoding from the higher order (HO) π-calculus. The main difference is that in HOπ the language does not contain an equality test for HO values. In πT there is such a test for data terms. This means that in the encoding the interior of a term must, so to speak, be open for inspection, and this makes our encoding and proof very different.

The applied π-calculus, the spi calculus [3], Burrows, Abadi and Needham's logic of authentication [9], which is nowadays known as BAN logic, and a number of other modelling and analysis techniques for security protocols rest on the Dolev-Yao assumption [11]. This means that the underlying crypto-system is considered unbreakable, so that the protocol logic on top of that is the only concern. A central aspect of all of these lines of work is that an analysis makes it necessary to explicitly represent the knowledge that an observer of a supposedly secure system can accumulate over time. This situation is similar in mobile process approaches: Observer knowledge has there been incorporated in notions of bisimulation and testing within the framework of the spi calculus; Borgström and Nestmann give a good overview of bisimulation for the spi calculus in [8]; Boreale, De Nicola and Pugliese have, besides treating bisimulation, shown how to take account of may-testing for the spi calculus in terms of knowledge-enriched traces [6]. In contrast to all of them our characterisation seems to be unique in that all observer knowledge is internalised. That is to say, it appears on the calculus level, not on the meta-level. In this sense it could be considered much closer to traditional process algebra methodology than the above-mentioned approaches.

6 Conclusion

We have presented a new encoding from the value-enriched mobile process calculus πT into the polyadic π-calculus. The cornerstone of the translation is an integrity manager that acts as a clearing house for all operations that involve translated values. It is protected by a firewall so that it cannot be impersonated by a hostile environment. Similar techniques were used in [2]. This encoding solves the long open full abstraction problem for πT-like calculi.

Our encoding is not compositional since it has the integrity manager and the firewall as a global context. While compositionality has been put forward as a criterion for whether an encoding from one calculus to the other is good [15], it can be argued (see [14]) that these criteria are too strong for practical purposes, and that by allowing a top-level context (but keeping the inner encoding compositional), many practically or theoretically motivated encodings turn out to be "good".

One way to interpret our result is that the π-calculus can provide as much "security" in the sense of protected data values as the πT-calculus and its instantiations such as the spi calculus. The means to achieve this are an integrity manager and a firewall. Thus we could claim to have proved that integrity management plus firewalling is a viable security philosophy. Modulo possible termonological differences this belief may actually be common; our contribution is to prove the correctness of a formal statement of it.

References

[1] M. Abadi and C. Fournet. Mobile Values, New Names, and Secure Communication. In *Principles of Programming Languages*, pages 104–115. ACM, 2001.

[2] M. Abadi, C. Fournet, and G. Georges. Secure Implementation of Channel Abstractions. In *Logic in Computer Science*, pages 105–116. IEEE, 1998.

[3] M. Abadi and A. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, 1999.

[4] M. Baldamus, J. Parrow, and B. Victor. Translating Spi Calculus to π-Calculus Preserving May-Tests. In *Logic in Computer Science*, pages 22–31. IEEE, 2004.

[5] M. Baldamus, J. Parrow, and B. Victor. A Fully Abstract Encoding of the π-Calculus with Data Terms. Technical Report 2005-004, Department of Information Technology, Uppsala University, February 2005.

[6] M. Boreale, R. De Nicola, and R. Pugliese. Proof Techniques for Cryptographic Processes. *SIAM Journal on Computing*, 31(3):947–986, 2002.

[7] J. Borgström, S. Briais, and U. Nestmann. Symbolic Bisimulation in the Spi Calculus. In *Concurrency Theory*, LNCS 3170, pages 161–176, 2004. Concur conference proceedings.

[8] J. Borgström and U. Nestmann. On Bisimulations for the Spi Calculus. Technical Report IC/2003/34, EPFL I&C, 2003.

[9] A. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989.

[10] R. De Nicola and M. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.

[11] D. Dolev and A. Yao. On the Security of Public-Key Protocols. *IEEE Transactions on Information Technology*, 29(2):198–208, 1983.

[12] R. Milner. Functions as Processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[13] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Parts I/II. *Information and Computation*, 100:1–77, 1992.

[14] U. Nestmann. What Is a 'Good' Encoding of Guarded Choice? *Information and Computation*, 156:287–319, 2000.

[15] C. Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous π-Calculus. In *Principles of Programming Languages*. ACM, 1997.

[16] D. Sangiorgi. From π-Calculus to Higher-Order π-Calculus – and Back. In *Theory and Practice of Software Development*, LNCS 668, pages 151–161. Springer, 1993.

[17] D. Sangiorgi and D. Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2003.