Preliminary Proceedings

5th International Workshop on

Security Issues in Concurrency (SecCo'07)

Lisbon, Portugal

September 3rd, 2007

Editors:

Daniele Gorla

Catuscia Palamidessi

Contents

Preface

This volume contains the proceedings of the Fifth Workshop on *Security Issues in Concurrency (SecCo'07)*. The workshop was held in Lisbon (Portugal) on September 3rd, 2007, as a satellite event to CONCUR'07. Previous editions of this workshops have been organized in Eindhoven (2003), London (2004) and San Francisco (2005); the 2006 edition in Bonn did not received enough submissions and so the meeting was not held.

The aim of the SecCo workshops is to cover the gap between the security and the concurrency communities. More precisely, the workshop promotes the exchange of ideas, trying to focus on common interests and stimulating discussions on central research questions. In particular, we called for papers dealing with security issues (such as authentication, integrity, privacy, confidentiality, access control, denial of service, service availability, safety aspects, fault tolerance, trust, language-based security) in emerging fields like web services, mobile ad-hoc networks, agent-based infrastructures, peer-to-peer systems, context-aware computing, global/ubiquitous/pervasive computing.

We received 13 submissions (including one short submission) and we have accepted 5 of them. The selection has been carried out by the program committee of SecCo'07, which consisted of

- Michael Backes (Saarland Univ., G);
- Tom Chothia (CWI, NL);
- Véronique Cortier (CNRS Loria, F);
- Yuxin Deng (Univ. of New South Wales, AUS);
- Daniele Gorla (Univ. "La Sapienza", IT) – co-chair;
- Heiko Mantel (RWTH, G);
- Mogens Nielsen (BRICS, DK);
- Flemming Nielson (DTU, DK);
- Catuscia Palamidessi (INRIA and Ecole polytechnique, F) – co-chair;
- Mark Ryan (Univ. of Birmingham, UK);
- Luca Viganò (Univ. Verona, IT);
- Jan Vitek (Purdue Univ., USA).

The papers were mostly refereed by the program committee and every paper received 3 or 4 reviews. In the reviewing phase, we have also been helped by some outside referees: Andy Brown, Stéphanie Delaune, Han Gao, Jun Pang, Henning Sudbrock, Terkel Tolstrup and Chenyi Zhang.

We had an invited talk shared with the EXPRESS'07 workshop that was given Cédric Fournet (Microsoft Research - Cambridge, UK) on "A Type Discipline for Authorization in Distributed Systems". We concluded the workshop with a panel discussion on "Information hiding: state-of-the-art and emerging trends". This was a venue where researchers from different areas of computer security presented common/orthogonal problems, techniques and goals related to information hiding. The panelists covered various aspects including data secrecy, anonymity, database security, information flow and protocol analysis; the approaches proposed ranged from language-based security to quantitative aspects and access control. The panelists were:

- Sabrina De Capitani di Vimercati (Univ. Milano, I);
- Steve Kremer (INRIA and ENS Cachan, F);
- Pasquale Malacaria (Queen Mary, UK);
- Peter Ryan (Newcastle Univ., UK);
- David Sands (Chalmers Univ., SE).

They presented their point of view on the topic and took questions from the audience.

We would like to thank whoever has contributed to SecCo'07. First of all, the program committee, the external reviewers, the invited speaker and the panelists. Then, we are very grateful to the CONCUR'07 workshop co-chairs, Antonio Ravara and Francisco Martins, for taking care of all the local organization and for managing the printing of these proceedings. We thanks the Elsevier Science B.V. (that will publish these proceedings electronically in the ENTCS series), and in particular Mike Mislove, Managing Editor of the ENTCS series. Last but not least, we are very grateful to all the authors that submitted a paper and to all the participants that attended the meeting.

Rome and Paris, July 27th, 2007

Daniele Gorla
Catuscia Palamidessi
SecCo'07 co-chairs

Invited Talk (joint with EXPRESS'07)

A Type Discipline for Authorization in Distributed Systems

Cédric Fournet[1]

Microsoft Research (Cambridge – UK)

Abstract

We consider the problem of statically verifying the conformance of the code of a system to an explicit authorization policy. In a distributed setting, some part of the system may be compromised, that is, some nodes of the system and their security credentials may be under the control of an attacker. To help predict and bound the impact of such partial compromise, we advocate logic-based policies that explicitly record dependencies between principals. We propose a conformance criterion, "safety despite compromised principals", such that an invalid authorization decision at an uncompromised node can arise only when nodes on which the decision logically depends are compromised. We formalize this criterion in the setting of a process calculus, and present a verification technique based on a type system. Hence, we can verify policy conformance of code that uses a wide range of the security mechanisms found in distributed systems, ranging from secure channels down to cryptographic primitives, including secure hashes, encryption, and public-key signatures.

[1] Joint work with Andrew Gordon and Sergio Maffeis.

Symbolic bisimulation for the applied pi calculus (extended abstract) ⋆

Stéphanie Delaune[a], Steve Kremer[b], Mark Ryan[c]

[a] *LORIA, CNRS & INRIA, Nancy, France*

[b] *LSV, CNRS & ENS de Cachan & INRIA, France*

[c] *School of Computer Science, University of Birmingham, UK*

Abstract

Recently, we have proposed in [10] a symbolic semantics together with a sound symbolic labelled bisimulation relation for the finite applied pi calculus. By treating inputs symbolically, our semantics avoids potentially infinite branching of execution trees due to inputs from the environment. This work is an important step towards automation of observational equivalence for the finite applied pi calculus, *e.g.* for verification of anonymity or strong secrecy properties. We present some of the difficulties we have encountered in the design of the symbolic semantics.

1 Introduction

The *applied pi calculus* [1] is a derivative of the pi calculus that is specialised for modelling cryptographic protocols. Participants in a protocol are modelled as processes, and the communication between them is modelled by means of channels, names and message passing. These messages are generated by a term algebra and equality is treated modulo an *equational theory*. For instance the equation $\mathsf{dec}(\mathsf{enc}(x, y), y) = x$ models the fact that encryption and decryption with the same key cancel out. *Active substitutions* model the availability of data to the environment.

As an example consider the following reduction step:

$$\nu s.k.\mathrm{out}(c, \mathsf{enc}(s, k)).P \xrightarrow{\nu x.out(c,x)} P \mid \{^{\mathsf{enc}(s,k)}/_{x}\}.$$

The process outputs on the channel c a secret name s encrypted with the key k. The active substitution $\{^{\mathsf{enc}(s,k)}/_{x}\}$ gives the environment the ability to access the

⋆ This work has been partly supported by the RNTL project POSÉ, EPSRC projects EP/E029833, *Verifying Properties in Electronic Voting Protocols* and EP/E040829/1, *Verifying anonymity and privacy properties of security protocols* and the ARTIST2 Network of Excellence. We thank Magnus Johansson and Björn Victor for interesting discussions.

term $\mathsf{enc}(s, k)$ via the fresh variable x without revealing either s, or k. The applied pi calculus generalizes the *spi calculus* [2] which only allows a fixed set of primitives built-in (symmetric and public-key encryption), while the applied pi calculus allows one to define these and other less usual primitives by means of an equational theory.

One of the difficulties in automating proofs in such a calculus is the infinite number of possible behaviours of the attacker, even in the case that the protocol process itself is finite. When the process requests an input from the environment, the attacker can give any term which can be constructed from the terms it has learned so far in the protocol, and therefore the execution tree of the process is potentially infinite-branching. To address this problem, researchers have proposed *symbolic abstractions* of processes, in which terms input from the environment are represented as symbolic variables, together with some constraints. These constraints describe the knowledge of the attacker (and therefore, the range of possible values of the symbolic variables) at the time the input was performed.

Reachability properties and also existence of off-line guessing attacks can be verified by solving the constraint systems arising from symbolic executions (e.g. [3,4]). *Observational equivalence properties* express the inability of the attacker to distinguish between two processes no matter how it interacts with them. These properties have been found useful for modelling anonymity and privacy properties (*e.g.* [9]), as well as other requirements [5,2]. Symbolic methods have already been used in the case of observational equivalence or bisimulation properties in classical process algebras (e.g. [11,6]). In particular, Borgström *et al.* [7] have defined a sound symbolic bisimulation for the spi calculus.

To show that a symbolic bisimulation implies the concrete one, we generally need to prove that the symbolic semantics is both sound and complete. Defining a symbolic semantics for the applied pi calculus has shown to be surprisingly difficult technically. In this paper, rather than describing our symbolic semantics we present several difficulties that we encountered and motivate some of our design choices. A complete description of the semantics and a sound symbolic bisimulation are available in [10].

2 Towards a symbolic semantics

In this section, we demonstrate some of the difficulties in defining a symbolic semantics for the applied pi calculus that is both sound and complete. Details about syntax and semantics of the original applied pi calculus are not given here. We will just recall some notions when we need them to explain the difficulties we have encountered.

2.1 Structural equivalence

The semantics of the applied-pi calculus is defined by structural rules defining three relations: structural equivalence ($\equiv$), internal ($\rightarrow$) and labelled ($\xrightarrow{\alpha}$) reduction. The last two relations are closed under structural equivalence. Hence, the natural first step seems to define a symbolic structural equivalence ($\equiv_s$) which is sound and complete in the following (informal) sense:

Soundness: $P_s \equiv_s Q_s$ implies for any valid instantiation σ, $P_s\sigma \equiv Q_s\sigma$;
Completeness: $P_s\sigma \equiv Q$ implies $\exists\, Q_s$ such that $P_s \equiv_s Q_s$ and $Q_s\sigma = Q$.

However, it seems difficult to achieve this. Before explaining the difficulty, we introduce structural equivalence more formally. *Structural equivalence* is the smallest equivalence relation $\equiv$ on extended processes that is closed under α-conversion on names and variables, application of evaluation contexts (an extended process with a hole), and some other standard rules such as associativity and and commutativity of the parallel operator and commutativity of the bindings. In addition the following three rules are related to active substitutions and equational theories.

$$\nu x.\{^M/_x\} \equiv 0 \qquad \{^M/_x\} \mid A \equiv \{^M/_x\} \mid A\{^M/_x\}$$

$$\{^M/_x\} \;\equiv\; \{^N/_x\} \quad \text{if } M =_{\mathsf{E}} N$$

Consider the process $P = \text{in}(c, x).\text{in}(c, y).\text{out}(c, f(x)).\text{out}(c, g(y))$ which can be reduced to $P' = \text{out}(c, f(M_1)).\text{out}(c, g(M_2))$ where M_1 and M_2 are two arbitrary terms provided by the environment. When $f(M_1) =_{\mathsf{E}} g(M_2)$, *i.e.* $f(M_1)$ and $g(M_2)$ are equal modulo the equational theory, we have that $P' \equiv \nu z.(\text{out}(c, z).\text{out}(c, z) \mid \{^{f(M_1)}/_z\})$, but this structural equivalence does not hold whenever $f(M_1) \neq_{\mathsf{E}} g(M_2)$.

The symbolic process $P'_s = \text{out}(c, f(x)).\text{out}(c, g(y))$ has to represent the different cases where $f(x)$ and $g(y)$ are equal or not. Hence, the question of whether the structural equivalence $P'_s \equiv_s \nu z.(\text{out}(c, z).\text{out}(c, z) \mid \{^{f(x)}/_z\})$ is valid cannot be decided, as it depends on the concrete values of x and y. Therefore, symbolic structural equivalence cannot be both sound and complete. This seems to be an inherent problem and it propagates to internal and labelled reduction, since they are closed under structural equivalence.

2.2 Intermediate semantics

The absence of sound and complete symbolic structural equivalence, mentioned above, significantly complicates the proof of our main result given in [10]. We therefore split it into two parts. We define a more restricted semantics which will provide an *intermediate* representation of applied pi calculus processes. These intermediate processes are a selected (but sufficient) subset of the original processes. One may think of them as being processes in some kind of normal form. They only have name restriction (no variable restriction) and all restrictions have to be in front of the process. They have to be *name and variable distinct* meaning that we have to use different names (resp. variables) to represent free and bound names (resp. variables), and also that any name (resp. variable) is at most bound once. Moreover, we require an intermediate process to be *applied* meaning that each variable in the domain of the process occurs only once. For instance, the process $A{\downarrow} = \nu b.\text{in}(c, y).\text{out}(a, f(b))$ is the intermediate process associated to the process $A = \nu x.(\text{in}(c, y).\nu b.\text{out}(a, x) \mid \{^{f(b)}/_x\})$.

Then, we equip these intermediate processes with a labelled bisimulation that coincides with the original one. The intermediate semantics differs from the semantics of the original applied pi calculus. In particular, we do not consider the three rules, given in Section 2.1, related to active substitution for structural equivalence and we do not substitute equals for equals in structural equivalence, but only in a controlled way in certain reduction rules. Thus, we avoid the problem mentioned

in the previous section.

Finally we present a symbolic semantics which is both sound and complete with respect to the intermediate one and give a sound symbolic bisimulation. Another way to tackle this problem may be to define a symbolic structural equivalence which is not complete and to prove directly completeness of the symbolic reduction and labeled transition without having completeness for structural equivalence, but we have not been able to carry this through to completion and therefore we did not take that approach.

2.3 Separate constraint systems and explicit renaming

To keep track of the constraints on symbolic variables we have chosen to associate a separate constraint system to each symbolic process. Keeping these constraint systems separate allows us to have a clean separation between the bisimulation and the constraint solving part. In particular we can directly build on existing work [4] and decide our symbolic bisimulation for a significant family of equational theories whenever the constraint system does not contain disequalities. This corresponds to the fragment of the applied pi calculus without else branches in the conditional. For this fragment, one may also notice that our symbolic semantics can be used to verify reachability properties using the constraint solving techniques from [8]. Another side-effect of the separation between the processes and the constraint system is that we forbid α-conversion on symbolic processes as we lose the scope of names in the constraint system. Thus, we have to deal with explicit renaming when necessary. This adds some additional complexity, but the benefit of keeping the constraints separate seems to be appealing in view of an implementation.

3 Conclusion and future work

Designing a symbolic semantics for the applied pi calculus has revealed to be much more difficult technically than expected. In this paper we have highlighted some difficulties we encountered and motivated some of our design choices. The result of these design choices is given in [10]. The obvious next step is to study solving constraint systems in the presence of disequalities. This would enable us to decide our bisimulation even in the case of else branches.

References

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proc. 28th Symposium on Principles of Programming Languages*, pages 104–115. ACM Press, 2001.

[2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proc. 4th Conference on Computer and Communications Security*, pages 36–47. ACM, 1997.

[3] R. Amadio, D. Lugiez, and V. Vanackère. On the symbolic reduction of processes with cryptographic functions. *Theoretical Computer Science*, 290:695–740, 2002.

[4] M. Baudet. *Sécurité des protocoles cryptographiques : aspects logiques et calculatoires.* Thèse de doctorat, Laboratoire Spécification et Vérification, ENS Cachan, France, 2007.

[5] B. Blanchet, M. Abadi, and C. Fournet. Automated Verification of Selected Equivalences for Security Protocols. In *Proc. 20th Symposium on Logic in Computer Science*, pages 331–340. IEEE Comp. Soc. Press, 2005.

[6] M. Boreale and R. D. Nicola. A symbolic semantics for the pi-calculus. *Information and Computation*, 126(1):34–52, 1996.

[7] J. Borgström, S. Briais, and U. Nestmann. Symbolic bisimulation in the spi calculus. In *Proc. 15th Int. Conference on Concurrency Theory*, volume 3170 of *LNCS*, pages 161–176. Springer, 2004.

[8] S. Delaune and F. Jacquemard. A decision procedure for the verification of security protocols with explicit destructors. In *Proc. 11th ACM Conference on Computer and Communications Security (CCS'04)*, pages 278–287. ACM Press, 2004.

[9] S. Delaune, S. Kremer, and M. D. Ryan. Coercion-resistance and receipt-freeness in electronic voting. In *Proc. 19th Computer Security Foundations Workshop*, pages 28–39. IEEE Comp. Soc. Press, 2006.

[10] S. Delaune, S. Kremer, and M. D. Ryan. Symbolic bisimulation for the applied pi calculus. Research Report LSV-07-14, Laboratoire Spécification et Vérification, ENS Cachan, France, Apr. 2007. 47 pages.

[11] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.

Detecting and Preventing Type flaws: a Control Flow Analysis with tags [1]

Chiara Bodei[1] Pierpaolo Degano[1] Han Gao[2] Linda Brodo[3]

[1] *Dipartimento di Informatica, Università di Pisa, Via Pontecorvo, I-56127 Pisa - Italia -* {chiara,degano}@di.unipi.it

[2] *Informatics and Mathematical Modelling, Technical University of Denmark, Richard Petersens Plads bldg 321, DK-2800 Kongens Lyngby - Denmark -* hg@imm.dtu.dk

[3] *Dipartimento di Scienze dei Linguaggi, Università di Sassari, via Tempio,9, I-07100 Sassari - Italia -* brodo@uniss.it

Abstract

A *type flaw attack* on a security protocol is an attack where an honest principal is cheated on interpreting a field in a message as the one with a type other than the intended one. In this paper, we shall present an extension of the LySA calculus with tags attached to each field, indicating the intended types. We developed a control flow analysis for analysing the extended LySA, which over-approximates all the possible behaviour of a protocol and hence is able to capture any type confusion that may happen during the protocol execution. The control flow analysis has been applied to a number of security protocols, either subject to type flaw attacks or not. The results show that it is able to capture type flaw attacks on those security protocols.

Keywords: Security Protocol, Control Flow Analysis, Type Flaw Attacks

1 Introduction

A *type flaw attack* on a security protocol arises when a field, originally intended to have one type, is instead interpreted as having another type. To prevent such attacks, the current techniques [11,12] consist in systematically associating each message field with a tag representing its intended type. Therefore fields with different types cannot be mixed up. Nevertheless, these may result in requiring extra and somehow unnecessary computational power and network transmission band. This is particularly heavy, when resources are limited such as in battery-powered embedded systems like PDAs, cell phones, laptops, etc.

In this paper, we explore these issues and propose a static analysis technique, based on Control Flow Analysis, for detecting potential type flaw attacks in the presence of a Dolev-Yao attacker [7]. The proposed approach abstracts the fields of protocol messages to a lower level, such that the misinterpretation can be formally

[1] This work has been partially supported by the project SENSORIA.

modeled. To this end, we extend the LySA calculus [2,3] with special tags, which represent the type of terms. The Control Flow Analysis approximates the behaviour of protocols in terms of the possibly exchanged messages and potential values of variables. The analysis can be working in either a *prescriptive* way, such that type flaws are avoided; or a *descriptive* way, such that type flaws are detected and recorded as violations of the intended types. Furthermore, if no type violation is found, we can prove that the protocol is free of type flaw attacks at run time. The analysis is fully automated and always terminates. It has been successfully applied to a number of protocols, such as Woo-Lam π_1[19] and Andrew Secure RPC [17].

LySA has been given different kinds of annotations for checking other security properties, e.g. confidentiality [9] and freshness [8]. It is very easy to combine tags with those techniques, thus giving a more comprehensive results of analysing security protocols.

The paper is organised as follows. In Section 2, we present the LySA calculus with tags for type flaw attacks, both the syntax and semantics are defined. We introduce the Control Flow Analysis in Section 3, which captures any type-mismatching that may happen. In Section 4, we show how the Control Flow Analysis works on two example protocols that are subject to type flaw attacks. In Section 5, we conclude with an assessment of our approach and a comparison with related work.

2 Calculus

The LySA calculus [2,3] is a process algebra, in the tradition of the π- [14] and Spi- [1] calculi. It differs from these essentially in two aspects. The first is the absence of channels: all processes have only access to a single global communication channel, the ether. The second aspect concerns the inclusion of pattern matching into the language constructs where values can become bound to values, i.e. into input and into decryption. This is different from having a separate matching construct, usually an if-then construct as in other process calculi and lead to more succinct specifications of protocols. We use here a dialect of LySA, which presents a more general pattern matching than the one in [2,3]. See also [5,16] for an alternative treatment.

Syntax of Terms

The basic blocks of LySA are values, used to represent agent names, nonces, keys. Syntactically, they are described by terms that may either be standard terms E or *matching terms* M. Standard terms – that can be names or variables – are used for modeling outputs and encryptions. Instead, for modeling inputs and decryptions we use matching terms, that, in turn, can be standard terms, or variables. We distinguish between *definition* (or binding) occurrences and *use* (or applied) occurrences of variables. A definition occurrence is when a variable gets its binding value, while an use occurrence is an appearance of a variable where its binding value is used.

The distinction is obtained by means of syntax: the definition occurrence of a variable x is denoted by $\natural x$, while in the scope of the declaration, the variable appears as x. Furthermore, this notation distinguishes variables from occurrences of standard terms in tuples of matching terms, by implicitly partitioning them into

standard terms or variables. In pattern matching, the first are checked for matching, while the others are bound in case of successful matching (see below).

$S ::=$ *standard terms*		$\mathcal{S} ::=$ *matching standard terms*	
n	name ($n \in \mathcal{N}$)	S	standard terms
x	use variable ($x \in \mathcal{X}_\mathcal{S}$)	$\natural x$	definition variable ($x \in \mathcal{X}_S$)

Here $\mathcal{N}$, $\mathcal{X}_S$, denote sets of names and of applied occurrences of variables, respectively. The name n is used to represent keys, nonces and names of principals.

Type Tagging We extend the syntax of standard LySA to cope with types, by using tags to represent the types of terms. Following [11], we assume to have a tag for each base type, such as *nonce*, *key*, etc. Moreover, we assume that the attacker is able to change only the types of terms that he can access. For malleability reasons, we choose to tag only encryptions and decryptions. In fact, by making the assumption of perfect cryptography, we have that only cleartext can be altered. Attackers can only forge an encryption when possessing the key used to cipher it.

$$\mathbf{Tag} \ni Tag ::= agent \mid nonce \mid key \mid \ldots$$

There are type variables, that are to standard variables such as tags are to closed terms (i.e. terms without variables). Similarly to the $\natural$-notation, we syntactically distinguish the defining occurrences of type variables (in the form $\sharp t$), from the corresponding use occurrences (in the form t). Syntactically, we have the following two new categories, where $\mathcal{X}_T$ denote sets of applied occurrences of type variables.

$T ::=$ *type terms*		$\mathcal{T} ::=$ *matching type terms*	
Tag	type tags ($Tag \in \mathbf{Tag}$)	T	type terms
t	use type variable ($t \in \mathcal{X}_\mathcal{T}$)	$\sharp t$	defining type variable ($t \in \mathcal{X}_\mathcal{T}$)

Furthermore, we can merge the above syntactic categories with the ones for standard terms in order to obtain the two more general syntactic categories for terms E and matching terms M. Encryptions are tuples of terms $E_1, \cdots, E_k$ encrypted under a term E_0 representing a shared key.

$E ::=$ *terms*		$M ::=$ *matching terms*	
S	standard terms	$\mathcal{S}$	matching standard terms
T	type terms	$\mathcal{T}$	matching type terms
$\{E_1, \cdots, E_k\}_{E_0}$	symmetric encryption ($k > 0$)		

We call Val the set of values, i.e. closed terms. Each value can have a type tag associated with it. From here on, for readability, we usually associate standard terms and type terms in encryptions and decryptions.

Syntax of Processes

In addition to the classical constructs for composing processes, our calculus also contains an input construct with matching and a decryption operation with matching. Furthermore, to keep track of the decryptions in which a violation occurs,

we decorate each decryption with a label l (from a numerable set $\mathcal{C}$). Labels are mechanically attached to program points in which decryptions occur (they are nodes in the abstract syntax tree of processes). Finally, by overloading the symbol ν, we use a new process construct to declare the expected type of a type variable.

$P ::=$ *processes*

$\langle E_1, \ldots, E_k \rangle.P$	output
$(M_1, \ldots, M_k).P$	input
decrypt E *as* $\{M_1, \ldots, M_k\}^l_{E_0}$ *in* P	decryption with matching
$(\nu\ n)P$	restriction
$(\nu\ \sharp t : Tag)P$	type declaration
$P_1 \mid P_2$	parallel composition
$!P$	replication
$\mathsf{0}$	nil

The sets of free variables, resp. free names, and of bound variables and names, of a term or a process are written $\mathsf{fv}(\cdot)$, $\mathsf{fn}(\cdot)$, $\mathsf{bv}(\cdot)$, $\mathsf{bn}(\cdot)$, respectively. they are defined in the standard way. As usual, we omit the trailing $\mathsf{0}$ of processes.

Our patterns – in the form $(M_1, \cdots, M_k)$ – are matched against tuples of terms $(E_1, \cdots, E_k)$. Note that, at run time, each $(E_1, \cdots, E_k)$ only includes closed terms, i.e. each variable composing each one of the E_i has been bound in the previous computations. Instead, matching terms M_i can be partitioned in closed terms and variables to be bound. Intuitively, the matching succeeds when the closed terms, say M_i, pairwise match to the corresponding terms E_i, and its effect is to bind the remaining terms E_j to the remaining variables $\natural x_j$. To exemplify, consider the following two processes, where only standard terms are present.

$$P = \mathit{decrypt}\ \{A, w_n\}_K\ \mathit{as}\ \{\natural x_a, N_B\}^{l_P}_K\ \mathit{in}\ P' \qquad Q = \mathit{decrypt}\ \{A, N_B\}_K\ \mathit{as}\ \{\natural x_a, \natural y_n\}^{l_Q}_K\ \mathit{in}\ Q'$$

The decryption in P succeeds only if $w_n = N_B$: in this case $\natural x_a$ will be bound to A. Instead, the second decryption in Q always succeeds, and results in binding $\natural x_a$ to A, and $\natural y_n$ to N_B.

The roles played by tags and type variables in the pattern matching are the same played by terms and variables. Suppose, e.g. to have the following processes:

$$\begin{aligned}
R = (\nu \sharp t_k : key)\mathit{decrypt}\ &\{(A, agent), (N_B, nonce), (z, key)\}_K\ \mathit{as} \\
&\{(A, agent), (N_B, nonce), (\natural z_k, \sharp t_k)\}^{l_R}_K\ \mathit{in}\ R' \\
\tilde{R} = (\nu \sharp t_k : key)\mathit{decrypt}\ &\{(A, agent), (N_B, nonce), (z, nonce)\}_K\ \mathit{as} \\
&\{(A, agent), (N_B, nonce), (\natural z_k, \sharp t_k)\}^{l_R}_K\ \mathit{in}\ R' \\
S = \mathit{decrypt}\ &\{(A, agent), (N_B, nonce), (z, t)\}_K\ \mathit{as} \\
&\{(A, agent), (N_B, nonce), (\natural z_k, key)\}^{l_S}_K\ \mathit{in}\ S'
\end{aligned}$$

The decryptions in R and $\tilde{R}$ always succeed and result in binding $\natural z_k$ to (the values assumed by) z, and $\sharp t_k$ to *key* or to *nonce*. In particular, in $\tilde{R}$ the decryption succeeds, even though the declared type for $\sharp t_k$ is *key*. In the decryption in S only if t successfully matches with *key* then $\natural z_k$ is bound to z.

Operational Semantics

Below we slightly modify the standard *structural congruence* $\equiv$ on LySA processes, also to take care of type declarations. It is the least congruence satisfying the following clauses:

- $P \equiv Q$ if P and Q are disciplined α-equivalent (as explained below);
- $(\mathcal{P}/_{\equiv}, |, \mathsf{0})$ is a commutative monoid;
- $(\nu n)\mathsf{0} \equiv \mathsf{0}, \quad (\nu n)(\nu n')P \equiv (\nu n')(\nu n)P, \quad (\nu n)(P \mid Q) \equiv P \mid (\nu n)Q \quad \text{if } n \notin \mathsf{fn}(P)$,
  $(\nu \sharp t : Tag)\mathsf{0} \equiv \mathsf{0}, \quad (\nu \sharp t : Tag)(\nu \sharp t' : Tag)P \equiv (\nu \sharp t' : Tag)(\nu \sharp t : Tag)P$,
  $(\nu \sharp t : Tag)(P \mid Q) \equiv P \mid (\nu \sharp t : Tag)Q \quad \text{if } \sharp t \notin \mathsf{bv}(P)$;
- $!P \equiv P \mid !P$

To simplify the definition of our control flow analysis in Section 3, we discipline the α-renaming of *bound* values and variables. To do it in a simple and "implicit" way, we assume that values and variables are "stable", i.e. that for each value $n \in \mathcal{N}$ there is a canonical representative $\lfloor n \rfloor$ for the set $\{n, n_0, n_1, \cdots\}$ and similarly, for each variable $x \in \mathcal{X}_S \cup \mathcal{X}_T$ there is a canonical representative $\lfloor x \rfloor$ for the set $\{x, x_0, x_1, \cdots\}$. Then, we discipline α-conversion as follows: two values (resp. variables) are α-convertible only when they have the same canonical value (resp. variable). In this way, we statically maintain the identity of values and variables that may be lost by freely applying α-conversions. Hereafter, we shall simply write n (resp. x) for $\lfloor n \rfloor$ (resp. $\lfloor x \rfloor$).

Following the tradition of the π-calculus, we shall give LySA a reduction semantics. The *reduction relation* $\rightarrow_{\mathcal{R}}$ is the least relation on closed processes that satisfies the rules in Table 1. It uses structural congruence, as defined above, and the disciplined treatment of α-conversion. We consider two variants of *reduction relation* $\rightarrow_{\mathcal{R}}$, graphically identified by a different instantiation of the relation $\mathcal{R}$, which decorates the transition relation. Both semantics use the type environment Γ, which maps a type variable in a set of tags.

$$\Gamma : \mathcal{X}_{\mathcal{T}} \rightarrow \wp(\mathbf{Tag})$$

One variant ($\rightarrow_{\mathsf{RM}}$) takes advantage of checks on type associations, while the other one ($\rightarrow$) discards them: essentially, the first semantics checks for type matching, while the other one does not (see below):

- the *reference monitor semantics* $\Gamma \vdash P \rightarrow_{\mathsf{RM}} Q$ takes

$$\mathcal{R}(E, M) = \begin{cases} \text{false} & \text{if } M = \sharp t \wedge E \notin \Gamma(\sharp t) \\ \text{true} & \text{otherwise} \end{cases}$$

  This function affects only type variables, i.e. only matching terms M in the form $\sharp t$. It checks whether the type $(\Gamma(\sharp t))$ associated with the variable includes E.

- the *standard semantics* $\Gamma \vdash P \to Q$ takes, by construction, $\mathcal{R}$ to be universally true (and therefore the index $\mathcal{R}$ is omitted).

Moreover, we define two auxiliary functions that handle the difference between closed terms and variables to be bound, by implicitly partitioning the tuples and treating the respective elements differently. We use a slightly modified notion of *substitution* applied to a process P, $P[E/M]$, where M can be either $\natural x$ or $\sharp t$.

$$P[E/M] = \begin{cases} P[M \mapsto E] \text{ if } M \in \{\natural x |\ x \in \mathcal{X}_S\} \cup \{\sharp t |\ t \in \mathcal{X}_T\} \\ P \qquad\qquad \text{otherwise} \end{cases}$$

The *pattern matching function* $comp(E, M)$ compares E against M only when M is a closed term and not a variable.

$$comp(E, M) = \begin{cases} \text{false} \ \text{ if } E \neq M \wedge \mathsf{fv}(M) \cup \mathsf{bv}(M) = \emptyset \\ \text{true} \ \text{ otherwise} \end{cases}$$

The judgement $\Gamma \vdash P \to_{\mathcal{R}} P'$ means that the process P can evolve into P', given the type environment Γ. The rule (Com) expresses that an output $\langle E_1, \ldots, E_k \rangle.P$

$$(Com)\quad \frac{\wedge_{i=1}^{k} comp(E_i, M_i)}{\Gamma \vdash \langle E_1, \ldots, E_k \rangle.P \mid (M_1, \ldots, M_k).Q \to_{\mathcal{R}} P \mid Q[E_1/M_1, \ldots, E_k/M_k]}$$

$$(Dec)\quad \frac{\wedge_{i=0}^{k} comp(E_i, M_i) \ \wedge \ \wedge_{i=1}^{k} \mathcal{R}(E_i, M_i)}{\Gamma \vdash \mathit{decrypt}\ \{E_1, \ldots, E_k\}_{E_0}\ \mathit{as}\ \{M_1, \ldots, M_k\}^{l}_{E_0}\ \mathit{in}\ P \to_{\mathcal{R}} P[E_1/M_1, \ldots, E_k/M_k]}$$

$$(Type\ Decl)\quad \frac{\Gamma[\sharp t \mapsto Tag] \vdash P \to_{\mathcal{R}} P'}{\Gamma \vdash (\nu\ \sharp t : Tag)P \to_{\mathcal{R}} (\nu\ \sharp t : Tag)P'} \qquad (Res)\quad \frac{\Gamma \vdash P \to_{\mathcal{R}} P'}{\Gamma \vdash (\nu\ n)P \to_{\mathcal{R}} (\nu\ n)P'}$$

$$(Par)\quad \frac{\Gamma \vdash P_1 \to_{\mathcal{R}} P_1'}{\Gamma \vdash P_1 \mid P_2 \to_{\mathcal{R}} P_1' \mid P_2} \qquad (Congr)\quad \frac{P \equiv P' \ \wedge \ \Gamma \vdash P' \to_{\mathcal{R}} P'' \wedge P'' \equiv P'''}{\Gamma \vdash P \to_{\mathcal{R}} P'''}$$

Table 1
Operational semantics, $\Gamma \vdash P \to_{\mathcal{R}} P'$, parameterised on $\mathcal{R}$.

is matched by an input $(M_1, \ldots, M_k)$ by checking whether the closed terms M_i are pairwise the same with the corresponding E_i (i.e. if $comp(E_i, M_i)$). When the matchings are successful, the remaining E_j are bound to the corresponding M_j (that are variables or type variables).

Similarly, the rule (Decr) expresses the result of matching an encryption $\{E_1, \ldots, E_k\}_{E_0}$ with $\mathsf{decrypt}\ E\ \mathsf{as}\{M_1, \ldots, M_k\}^{l}_{E_0'}\ \mathsf{in}\ P$. As it was the case for communication, the closed terms M_i must match with the corresponding E_i, and ad-

ditionally the keys must be the same. When the matching is successful the remaining terms E_j are bound to the corresponding M_i (that are definition variables or definition type variables). Recall that in the *reference monitor semantics* we ensure that the components of the decrypted message have the types expected, by checking whether the $\sharp t$ are bound to a type tag that is included in $\Gamma(\sharp t)$. In the *standard semantics* the condition $\mathcal{R}(E, M)$ is universally true and thus can be ignored. Back to our example processes R, $\tilde{R}$, S, we have that in R, $comp(z, \natural z_k) = comp(key, \sharp t) = true$ and $\mathcal{R}(key, \sharp t) = true$ (because $key \in \Gamma(\sharp t)$), while in $\tilde{R}$, $comp(z, \natural z_k) = comp(nonce, \sharp t) = true$, but $\mathcal{R}(nonce, \sharp t) = false$ (because $nonce \notin \Gamma(\sharp t)$). Note also that in S, $comp(t, key) = true$ only if $t = key$, and, in this case $P[z/\natural z_k] = P[\natural z_k \mapsto z]$.

The rule (Type Decl) records the new association between the type variable $\sharp t$ and the type Tag in the type environment Γ. The updating of Γ is indicated as $\Gamma[\sharp t \mapsto Tag]$.

The rules (Repl), (Par) and (Congr) are standard.

Dynamic Property

As for the dynamic property of the process, we shall consider a process free of type flaw attack, when in all computations, each type variable is bound to the expected type. Consequently, the reference monitor will never stop any execution step. Note that we only consider the type flaws occurring inside encryptions and decrytpions.

Definition 2.1 A process P is *free of type flaw attacks* when for each step $\Gamma \vdash P \rightarrow P'$, we always have $\Gamma \vdash P \rightarrow_{\mathsf{RM}} P'$.

3 Static Analysis

We develop a control flow analysis for analysing tagged LySA processes. The aim of the analysis is to safely over-approximate all the possible protocol behaviour which permits to safely approximate when the reference monitor may abort the computation of a process P. The approximation is represented by a tuple $(\Gamma, \rho, \kappa, \psi)$ (resp. a pair (ρ, ϑ) when analysing a term E), called *estimate* for P (resp. for E), that satisfies the judgements defined by the axioms and rules of Table 2. In particular, the analysis records which value tuples may flow over the network and which values may be bound to each *definition variable* (e.g. $\natural x$) and *definition type variable* (e.g. $\sharp t$). Moreover, at each decryption place, the analysis checks whether a *type tag* (e.g. Tag) bound to each *definition type variable* is the intended one, or a violation is reported. The analysis is defined in the flavor of Flow Logic [15].

Analysis of Terms

The judgement for analysing terms is $\rho \models E : \vartheta$. The analysis keeps track of the potential values of *variables* or *type variables*, e.g. x or t, by recording them into the global *abstract environment* ρ:

- $\rho : \mathcal{X}_{\mathcal{S}} \cup \mathcal{X}_{\mathcal{T}} \rightarrow \wp(Val)$ maps variables and type variable to the sets of values that they may be bound to.

The judgement is defined by the axioms and rules in the upper part of Table 2. Basically, the rules amount to demanding that ϑ contains all the values associated with the components of a term, e.g. a name n evaluates to the set ϑ, provided that n belongs to ϑ; similarly for a variable x, provided that ϑ includes the set of values $\rho(x)$ to which x is associated with.

Analysis of Processes

In the analysis of processes, the information on the possible values, that may flow over the network, is collected into the component κ:

- $\kappa \subseteq \wp(Val^*)$: the *abstract network environment* that includes all the value-tuples forming a message that may flow on the network.

The judgement for processes takes the form: $\rho, \kappa, \Gamma \models P : \psi$, where the components ρ, κ, and Γ are as above (recall that $\Gamma : \mathcal{X}_T \to \wp(\mathbf{Tag})$), while $\psi \subseteq \mathcal{C}$, is the (possibly empty) set of "error messages" of the form l, indicating that a type-mismatching (or violation) may happen at the decryption, labelled l. The judgement is defined by the axioms and rules in the lower part of Table 2 (where $X \Rightarrow Y$ means that Y is only evaluated when X is *True*) and are explained later.

Before commenting on the analysis rules, we introduce three auxiliary functions, all of which generate some logic formulas to be used in the analysis rules. See some examples below.

The first one is the *matching* function, which takes care of pattern matching a value v to a matching term M. Remember that pattern matching cannot be performed on either $\natural x$ or $\sharp t$, requiring that M has to be some S or T. If this is the case, matching succeeds when v is an evaluation of the value of S or T.

$$match(v, M, \rho) = \begin{cases} \text{false} & \text{if } M \in \{S, T\} \wedge v \notin \vartheta \text{ where } \vartheta \text{ is s.t. } (\rho \models M : \vartheta) \\ \text{true} & \text{otherwise} \end{cases}$$

The second one is a *substitution* function, which corresponds to the notion of variable binding. Intuitively, it only makes sense to bind a value to either a *definition variable* or a *definition type variable*. So the substitution function binds the value v to M only when M is variable $\natural x$ or a type variable $\sharp t$.

$$sub(v, M) = \begin{cases} \text{false if } v \notin \rho(M) \text{ with } M \in \{\natural x |\ x \in \mathcal{X}_S\} \cup \{\sharp t |\ t \in \mathcal{X}_T\} \\ \text{true otherwise} \end{cases}$$

The last function is about *type checking*. Given a type environment Γ, it checks whether v is the expected type of a *definition type variable* $\sharp t$. If it is not the case, the decryption labeled l, is recorded in the error component ψ. Note that in order to let the type checking work, M has to be a definition type.

$$chk(v, M, \Gamma, l, \psi) = \begin{cases} (v \neq \Gamma(\sharp t) \Rightarrow l \in \psi) & \text{if } M \in \{\sharp t |\ t \in \mathcal{X}_T\} \\ \text{true} & \text{otherwise} \end{cases}$$

$match(m,n)$	$= (\rho \models n : \vartheta \ \wedge \ m \in \vartheta)$	$match(m, \natural x)$	$= true$
$sub(m, \natural x)$	$= (m \in \rho(x))$	$sub(m,n)$	$= true$
$chk(m, \sharp t, \Gamma, l)$	$= (m \neq \Gamma(t) \Rightarrow l \in \psi)$	$chk(m,n,\Gamma,l)$	$= true$

$$(Const)\ \frac{N \in \vartheta}{\rho \models N : \vartheta}\ (N = Tag \text{ or } n) \qquad (Var)\ \frac{\rho(X) \subseteq \vartheta}{\rho \models X : \vartheta}\ (X = x \text{ or } t)$$

$$(Encr)\ \frac{\begin{array}{c}\wedge_{i=0}^{k}\ \rho \models E_i : \vartheta_i\ \wedge \\ \forall v_0, \ldots, v_k : \wedge_{i=0}^{k}\ v_i \in \vartheta_i \Rightarrow \{v_1, \ldots, v_k\}_{v_0} \in \vartheta\end{array}}{\rho \models \{E_1, \ldots, E_k\}_{E_0} : \vartheta}$$

$$(Out)\ \frac{\begin{array}{c}\wedge_{i=1}^{k}\ \rho \models E_i : \vartheta_i\ \wedge \\ \forall v_1, \ldots, v_k : \wedge_{i=1}^{k} v_i \in \vartheta_i \Rightarrow \langle v_1, \ldots, v_k\rangle \in \kappa\ \wedge \\ \rho, \kappa, \Gamma \models P : \psi\end{array}}{\rho, \kappa, \Gamma \models \langle E_1, \ldots, E_k\rangle.P : \psi}$$

$$(In)\ \frac{\begin{array}{c}\forall \langle v_1, \ldots, v_k\rangle \in \kappa\ \ \wedge_{i=1}^{k}\ (match(v_i, M_i) \Rightarrow sub(v_i, M_i))\ \wedge \\ \rho, \kappa, \Gamma \models P : \psi\end{array}}{\rho, \kappa, \Gamma \models (M_1, \ldots, M_k).P : \psi}$$

$$(Dec)\ \frac{\begin{array}{c}\rho \models E : \vartheta\ \ \wedge\ \ \rho \models E_0 : \vartheta_0\ \wedge \\ \forall \{v_1, \ldots, v_k\}_{v_0} \in \vartheta : v_0 \in \vartheta_0 \Rightarrow \\ \wedge_{i=1}^{k}(match(v_i, M_i) \Rightarrow (sub(v_i, M_i)\ \ \wedge\ \ chk(v_i, M_i, \Gamma, l)))\ \ \wedge \\ \rho, \kappa, \Gamma \models P : \psi\end{array}}{\rho, \kappa, \Gamma \models decrypt\ E\ as\ \{M_1, \ldots, M_k\}^{l}_{E_0}\ in\ P : \psi}$$

$$(TNew)\ \frac{(\sharp t, Tag) \in \Gamma\ \ \wedge\ \ \rho, \kappa, \Gamma \models P : \psi}{\rho, \kappa, \Gamma \models (\nu\ \sharp t : Tag)P : \psi} \qquad (Par)\ \frac{\rho, \kappa, \Gamma \models P_1 : \psi\ \ \wedge\ \ \rho, \kappa, \Gamma \models P_2 : \psi}{\rho, \kappa, \Gamma \models P_1 \mid P_2 : \psi}$$

$$(Res)\ \frac{\rho, \kappa, \Gamma \models P : \psi}{\rho, \kappa, \Gamma \models (\nu\ n)P : \psi} \qquad (Rep)\ \frac{\rho, \kappa, \Gamma \models P : \psi}{\rho, \kappa, \Gamma \models !P : \psi} \qquad (Nil)\ \ \rho, \kappa, \Gamma \models 0 : \psi$$

Table 2
Analysis of tagged Lysa Terms: $\rho \models E : \vartheta$, and Processes: $\rho, \kappa, \Gamma \models P : \psi$

We now briefly comment on the rules for analysing processes. In the premises of the rule for k-ary *output* (Out), we require that all the terms are abstractly evaluated, and that all the combinations of these values are recorded in κ, since they are the values that may be communicated. Finally, the continuation process must be analysed.

The rule (In) describes the analysis of pattern matching *input* and uses both the *match* function and *substitution*. The idea is to examine all the sequences of $\langle v_1, ..., v_k\rangle$ in the κ component and to point-wise compare it against the tuple of matching terms $(M_1, ..., M_k)$. The *matching* function selects only the closed terms

and for each of them, say M_i, checks whether the corresponding v_i is included in ϑ_i, i.e. the result of the analysis for M_i. If the matching succeeds for all the closed terms, then, the substitution function takes care of binding the remaining values v_j to the corresponding *definition variables* or *definition type variables* M_j. Moreover, the continuation process must be analysed.

The rule for *decryption* (Dec) is quite similar to the rule for *input*: matching and substitution are handled in the same way. The values to be matched are those obtained by evaluating the term E and the matching ones are the terms inside the decryption. If the matching succeeds for all closed terms, then the substitution is applied to the remaining values that are bound to the corresponding definition variables or definition type variables. When processing the *substitution*, *type checking* is also performed to capture violations. These occur when a *definition type variable* is bound to an unexpected type. In this case, the label l of the decryption is recorded in the error component ψ. Both in the case of input and decryption we make sure only to analyse the continuation process P in those cases where the input or decryption could indeed succeed.

The rule for *type declaration* (TNew) requires that the declared type is recorded in the type environment Γ.

The rule for the *inactive process* (Nil) does not restrict the analysis result, while the rules for *parallel composition* (Par), *restriction* (Res), and *replication* (Rep) ensure that the analysis also holds for the immediate subprocesses.

Semantic properties

Our analysis is semantically correct regardless of the way the semantics of LySA is parameterised (see [4] for the formal proofs). More precisely, we proved a subject reduction theorem for both the standard and the reference monitor semantics: if $(\rho, \kappa, \Gamma) \models P : \psi$, then the same tuple $(\rho, \kappa, \psi, \Gamma)$ is a valid estimate for all the states passed through in a computation of P, i.e. for all the derivatives of P.

Theorem 3.1 (Subject reduction) *If* $\Gamma \vdash P \to Q$ *and* $\rho, \kappa, \Gamma \models P : \psi$ *then also* $\rho, \kappa, \Gamma \models Q : \psi$. *Furthermore, if* $\psi = \emptyset$ *then* $P \to_{RM} Q$

In addition, when analysing a process P if the error component ψ is empty then the reference monitor *cannot stop* the execution of P. This means that our analysis correctly predicts when we can safely do without the reference monitor.

Theorem 3.2 (Static check for reference monitor) *If* $\rho, \kappa, \Gamma \models P : \psi$ *and* $\psi = \emptyset$ *then RM cannot abort* P.

Example

Consider a scenario in which a principal A sends out an encrypted nonce onto the network and another principal B is expecting an encrypted key receiving from the network. Assume both encryptions use the same key K, obviously, B could be cheated on accepting the nonce as the key.

$$\begin{aligned} A \to \quad & : \{N\}_K \\ \to B & : \{K'\}_K \end{aligned}$$

Our control flow analysis can work in two ways depending on how the protocol is modelled: either detecting what B received is a wrong one or preventing B from accepting it.

• In case the goal is to ***detect*** any type flaw attack may happen to the protocol, we can model it as follows,

$$\langle A, \{(N, nonce)\}_K \rangle.0$$
$$\mid (\nu\, \sharp tx_n : key)\ \ (A, \natural x_{enc}).\ \ decrypt\ x_{enc}\ as\ \{(\natural x_n, \sharp tx_n)\}^l_K\ in\ 0$$

where the type of the encrypted message that B received, i.e. $\sharp tx_n$, is declared to be key. The analysis then gives rise to the analysis components ρ, κ, Γ and ψ with the following entries:

$$\langle A, \{(N, nonce)\}_K \rangle \in \kappa \quad (\sharp tx_n, key) \in \Gamma \quad l \in \psi$$
$$\{(N, nonce)\}_K \in \rho(x_{enc}) \quad N \in \rho(x_n) \qquad nonce \in \rho(tx_n)$$

which show that the attack is captured by $l \in \psi$

• In case one wants to ***prevent*** such a type flaw attack from happening, the protocol can be modelled as,

$$\langle A, \{(N, nonce)\}_K \rangle.0$$
$$\mid (A, \natural x_{enc}).\ \ decrypt\ x_{enc}\ as\ \{(\natural x_n, key)\}^l_K\ in\ 0$$

It requires that the message inside the encryption that B got has to be a key. In this case, the analysis result becomes:

$$\langle A, \{(N, nonce)\}_K \rangle \in \kappa \quad \Gamma = \emptyset \qquad \psi = \emptyset$$
$$\{(N, nonce)\}_K \in \rho(x_{enc}) \quad \rho(x_n) = \emptyset$$

Now $\rho(x_n) = \emptyset$ shows that no value binds to the variable x_n, i.e. the type flaw attack is successfully prevented.

Modelling the Attacker

In our work, the protocol and the attacker are formally modelled as two parallel processes, $P_{sys} \mid P_\bullet$, where P_{sys} represents the protocol process and $P_\bullet$ is some arbitrary attacker. The attacker considered here is the Dolev-Yao attacker [7], who is an active attacker and assumed to have the overall control of the network, over which principals exchange messages. Therefore he has access to messages transmitted over the network and is able to eavesdrop or replay messages sending over the network but also to encrypt, decrypt or generate messages provided that the necessary information is within his knowledge. Instead, secret messages and keys, e.g. (νK_{AB}), are restricted to their scope in P_{sys} and thus not immediately accessible to the attacker. To deal with types, we require that the attacker is able to change types of terms that are accessible to him. Due to space limitations, we shall not go further into details here, rather we refer to [3] for a description about modelling the attacker in a similar setup, as well as for a similar treatment of semantic correctness.

4 Validation

To verify the usefulness of our Control Flow Analysis, a number of experiments have been performed on security protocols from the literature. In this section, we shall show the analysis results of some example protocols, which are subject to type flaw attacks, namely the Woo and Lam protocol, version π_1 and the Andrew Secure RPC protocol (both the original version and the BAN version with type flaw corrected). The analysis results show that those type flaw attacks are successfully captured. Furthermore, it proves that after BAN's correction, the Andrew Secure RPC protocol does not suffer from type flaw attacks any longer.

Woo and Lam Protocol π_1

Woo and Lam [19] introduced a protocol that ensures one-way authentication of the initiator of the protocol, A, to a responder, B. The protocol uses symmetric-key cryptography and a trusted third-party server, S, with whom A and B share long-term symmetric keys. The protocol uses a fresh nonce N_B produced by B. The protocol narration is listed in the left part of the figure below, where K_{AS} and K_{BS} represent the long-term keys that A and B share with the trusted server S.

1. $A \rightarrow B : A$	1. $M(A) \rightarrow B : A$
2. $B \rightarrow A : N_B$	2. $B \rightarrow M(A) : N_B$
3. $A \rightarrow B : \{A, B, N_B\}_{K_{AS}}$	3. $M(A) \rightarrow B : N_B$
4. $B \rightarrow S : \{A, B, \{A, B, N_B\}_{K_{AS}}\}_{K_{BS}}$	4. $B \rightarrow M(S) : \{A, B, N_B\}_{K_{BS}}$
5. $S \rightarrow B : \{A, B, N_B\}_{K_{BS}}$	5. $M(S) \rightarrow B : \{A, B, N_B\}_{K_{BS}}$
the protocol narration	*the type flaw attack*

The Woo-Lam protocol is prown to a type flaw attack, which is shown in the right part of the figure. The attacker replays the nonce N_B to B in step 3, which B accepts as being of the form $\{A, B, N_B\}_{K_{AS}}$. B then encrypts whatever he received and then sends it out in step 4. The attacker intercepts it and replays it to B in step 5 and therefore fools B to believe that he has authenticated A, whereas A has not even participated in the run.

In LySA, the Woo-Lam protocol is modelled as three processes, A, B and S, running in parallel within the scope of the shared keys, say $P_{WL} = (\nu\ K_{AS})(\nu\ K_{BS})(A \mid B \mid S)$, each of which represents the sequence of actions of one principal as listed below. For clarity, each message begins with the pair of principals involved in the exchange.

$$
\begin{aligned}
&\text{Principal } A : \qquad (\nu\ \sharp tx_{nb} : nonce)\\
&\qquad /*1*/\ \langle A, B, A\rangle.\\
&\qquad /*2*/\ (B, A, (\natural x_{nb}, \sharp tx_{nb})).\\
&\qquad /*3*/\ \langle A, B, (\{A, B, (x_{nb}, tx_{nb})\}_{K_{AS}}, \{agent, agent, nonce\}_{key})\rangle.0
\end{aligned}
$$

$$
\begin{array}{rl}
\text{Principal } B: & /*1*/\ (A,B,A).\\
& /*2*/\ (\nu\ N_B)\ \ \langle B,A,(N_B,nonce)\rangle.\\
& /*3*/\ (A,B,(\natural ya_{enc},\sharp tya_{enc})).\\
& /*4*/\ \langle B,S,\{A,B,(ya_{enc},tya_{enc})\}_{K_{BS}}\rangle.\\
& /*5*/\ (S,B,(\natural ys_{enc},\sharp tys_{enc})).\\
& \qquad\quad decrypt\ ys_{enc}\ as\ \{A,B,(N_B,nonce)\}^{l_1}_{K_{BS}}\ in\ 0\\
\text{Server } S: & \qquad\quad (\nu\ \sharp tza_{enc}:enc)\ (\nu\ \sharp tz_{nb}:nonce)\\
& /*4*/\ (B,S,(\natural zy_{enc},\sharp tzy_{enc})).\\
& \qquad\quad decrypt\ zy_{enc}\ as\ \{A,B,(\natural za_{enc},\sharp tza_{enc})\}^{l_2}_{K_{BS}}\ in\\
& \qquad\quad decrypt\ za_{enc}\ as\ \{A,B,(\natural z_{nb},\sharp tz_{nb})\}^{l_3}_{K_{AS}}\ in\\
& /*5*/\ \langle S,B,\{A,B,(z_{nb},tz_{nb})\}_{K_{BS}}\rangle.0
\end{array}
$$

For the Woo and Lam protocol, we have $(\rho,\kappa,\Gamma)\models P_{WL}:\psi$, where ρ, κ and Γ have the following non-empty entries (we only list here the interesting ones):

$$\rho(za_{enc})=\{\{A,B,(N_B,nonce)\}_{K_{AS}},N_B\}\qquad(\sharp tza_{enc},enc)\in\Gamma$$

$$\rho(tza_{end})=\{\{agent,agent,nonce\}_{key},nonce\}\ \{l_2\}\in\psi$$

The error component has a non-empty set, $\psi=\{l_2\}$, showing that a violation may happen in the decryption marked with label l_2 (the second line of step 4 in S). This is the place where S is trying to decrypt and bind values to the variable za_{enc} and its type variable tza_{enc}, which, as indicated by Γ, can only be $\{A,B,(N_B,nonce)\}_{K_{AS}}$. However, $\rho(za_{enc})$ and $\rho(tza_{enc})$ suggest that za_{enc} may also have the value N_B and tza_{enc} may have the value *nonce*. This violates the type assertion and amounts to the fact that, in step 4, S receives the message $\{A,B,N_B\}_{K_{BS}}$ instead of the expected one $\{A,B,\{A,B,N_B\}_{K_{AS}}\}_{K_{BS}}$. This exactly corresponds to the type flaw shown before.

Andrew Secure RPC protocol

The goal of the Andrew Secure RPC protocol is to exchange a fresh, authenticated, secret key between two principals sharing a symmetric key K. In the first message, the initiator A sends a nonce N_A, the responder B increments and returns it as the second message together with his nonce N_B. A accepts the value and returns the N_B+1, B receives and checks the third message and if it contains the nonce incremented, then he sends a new session key, K' to A together with a new value N'_B to be used in subsequent communications.

1. $A\to B: A,\{N_A\}_K$	1. $A\to B:\quad A,\{N_A\}_K$
2. $B\to A: \{N_A+1,N_B\}_K$	2. $B\to A:\quad \{N_A+1,N_B\}_K$
3. $A\to B: \{N_B+1\}_K$	3. $A\to M(B): \{N_B+1\}_K$
4. $B\to A: \{K',N'_B\}_K$	4. $M(B)\to A: \{N_A+1,N_B\}_K$
the protocol narration	*the type flaw attack*

Also, the Andrew Secure RPC protocol [17] is subject to type flaw attack as shown above in the right part of the figure: by replaying the message from step 2 to B in step 4, the attacker can successfully force A to accept $N_A + 1$ as the new session key. The protocol makes use of an operation to increment N_A, in step 2, and N_B, in the third step (see [3] for the possible model of SUCC)).

The protocol can be modelled as $P_{Andrew} = (\nu\ K)(A \mid B)$, where K is the shared key and A and B are defined as follows (we only list the relevant steps).

$$
\begin{array}{ll}
\text{Principal } A: & (\nu\ N_A)\ \ (\nu\ \sharp tx_k : key)\ \ (\nu\ \sharp tx_{nb'} : nonce) \\
& /*1*/\ \langle A, B, A, \{(N_A, nonce)\}_K\rangle.\ldots. \\
& /*4*/\ (B, A, \natural x_{enc}). \\
& \qquad decrypt\ x_{enc}\ as\ \{(\natural x_k, \sharp tx_k), (\natural x_{nb'}, \sharp tx_{nb'})\}_K^{l_{x1}}\ in\ 0 \\
\text{Principal } B: & (\nu\ N_B)(\nu\ N'_B)(\nu\ K')(\nu\ \sharp ty_{na} : nonce) \\
& /*1*/\ (A, B, A, \natural y_{enc}). \\
& \qquad decrypt\ y_{enc}\ as\ \{(\natural y_{na}, \sharp ty_{na})\}_K^{l_{y1}}\ in \\
& /*2*/\ \langle B, A, \{(y_{na}+1, ty_{na}), (N_B, nonce)\}_K\rangle.\ldots.
\end{array}
$$

For the Andrew Secure RPC protocol, we have $(\rho, \kappa, \Gamma) \models P_{Andrew} : \psi$, where ρ, κ and Γ have the following non-empty entries (we only list here the interesting ones):

$$
\begin{array}{ll}
\langle B, A, \{(N_A+1, nonce), (N_B, nonce)\}_K\rangle \in \kappa & \\
\rho(x_k) = \{K', N_a + 1\} & \rho(tx_k) = \{key, nonce\} \\
(\sharp tx_k, key) \in \Gamma & (l_{x1}) \in \psi
\end{array}
$$

The component κ collects all the messages potentially flowing over the network, including the one sent by B in step 2, namely $\langle B, A, \{(N_A+1, nonce), (N_B, nonce)\}_K\rangle$. This message could be received by A in his fourth step (e.g. replayed by an attacker) and consequently binding $N_A + 1$ to $\natural x_k$ and $nonce$ to $\sharp tx_k$, which can be verified by examining the content of ρ (i.e. $N_A + 1 \in \rho(x_k)$ and $nonce \in \rho(tx_k)$). However, as suggested by Γ, the expected type of the type variable tx_k can *only* be *key* (by $\Gamma(tx_k) = \{key\}$) but not *nonce*. This violation is captured by the analysis by recording the label l_{x1} in the error component ψ (by $l_{x1} \in \psi$).

Andrew Secure RPC protocol with type flaw corrected

An improved version of Andrew Secure RPC protocol is suggested in [6] in order to prevent the above mentioned type flaw attack. The fixing amounts to inserting another component N_A into the encryption in the fourth message, as shown below,

$$4'.\ \ B \to A: \ \ \{K', N'_B, N_A\}_K$$

Now the encryption in step 2 has two fields and in step $4'$, A is expecting an encryption of 3 fields, therefore the attacker is no longer able to replay the message from step 2 and consequently make A accept nonce as a fresh key. This claim is verified by applying our analysis, which gives an empty error component, i.e. $\psi = \emptyset$.

5 Conclusion and Related Work

A type flaw attack happens when a field in a message is interpreted as having a type other than the originally intended one. In this paper, we extended the syntax of the process calculus, LySA, with tags, which represent the intended types of terms. The semantics of the tagged LySA makes use of a *reference monitor* to capture type-mismatching at run time.

On the static side, we developed a control flow analysis for the tagged LySA processes to check at each decryption place that whether the received, secret data has the right type. The static analysis ensures that, if each component of an encryption received by a principal is of the intended type, then the process is not subject to a type flaw attack at execution time. Actually, for malleability reasons, we only consider type flaws attacks occurring inside encryptions and decryptions. As far as the attacker is concerned, we adopted the notion from Dolev-Yao threat model and extended it with tags in order to fit it into our setting. The control flow analysis has been applied to a number of protocols, e.g. Woo-Lam π_1 and Andrew Secure RPC as shown in Section 4, and has confirmed that we can successfully detect type flaw attacks on the protocols.

Type flaw attacks on security protocols have been studied for some years, e.g. [11] also adopted the technique of tagging each message field with intended type, and later on, [12] simplified the tag structure for encryption. However these works aim at preventing type flaw attacks in the protocol execution stage by attaching some extra bits, representing types, to the messages transmitted over the network, and consequently the size of each message is increased, which results in raising unnecessary burden to the underlying network. Other works on type flaw attacks include applying type and effect system to security protocols, e.g. [10], such that a protocol is free of type flaw attacks if it is type checked. Type Systems are normally prescriptive(i.e. they infer types and impose the well-formedness conditions at the same time), while Control Flow Analysis is normally descriptive (i.e. it merely infers the information and then leave it to a separate step to actually impose demands on when programs are well-formed). Our approach offers a mix of both ways. Indeed, it can be either *descriptive*, i.e. it describes when the protocol does not respect the typing (via binding of type variables) or *prescriptive*, i.e. some flaws are avoided (via matching of tag terms). Under this regard, launching the tool implementing our analysis can then correspond to a sort of approximate type checking. More specifically, our control flow analysis can be used to 1) detect type flaw attacks: it can be applied in the protocol design stage: once a tagged protocol process is analyzed to be free of type flaw attacks, it can be used untagged while still ensures security; or 2) prevent type flaw attacks: the tags work in a way such that fields with different types cannot be mixed up. Therefore, it offers flexibility in satisfying different needs.

LySA has been developed to be decorated by several kinds of annotations and successfully applied for checking different security properties, e.g. confidentiality [9] and freshness [8]. It is very easy to combine tags with those techniques, thus obtaining a more general form of analysis. The core analysis can remain the same: different inspections of a solution permit to check different security properties of a

protocol, with no need of re-analysing it several times.

The control flow analysis presented here is designed to capture *simple* type flaw attacks, i.e. one field is confused with another single field. Future work will extends the analysis to deal with more *complex* ones [18], as considered in [13], e.g. when a single field in a message is confused with a concatenation of fields. Furthermore, we can think about more complex kinds of tags.

Acknowledgments. We are grateful to Hanne Riis Nielson and Terkel K. Tolstrup for their helpful discussions and comments.

References

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

[2] C. Bodei, M. Buchholtz, P. Degano, F. Nielson and H.R. Nielson. Automatic Valication of Protocol Narration. *In Proc. of CSFW'03*, IEEE Press.

[3] C. Bodei, M. Buchholtz, P. Degano, F. Nielson and H.R. Nielson. Static Validation of Security Protocols. *Journal of Computer Security*, 13(3), pp.347 - 390, 2005.

[4] C. Bodei, P. Degano, H. Gao, L. Brodo. Detecting and Preventing Type flaws: a Control Flow Analysis with tags. TR-07-16, Dipartimento di Informatica, Università di Pisa, 2007.

[5] M. Buchholtz, F. Nielson and H.R. Nielson. A Calculus for Control Flow Analysis of Security Protocols. *International Journal of Information Security*, 2(3-4), pp.145-167, 2004.

[6] M. Burrows and M. Abadi and R. Needham. A Logic of Authentication. ACM. *Transactions in Computer Systems*, 8(1), pp. 18-36, 1990.

[7] D. Dolev and A.C. Yao. On the Security of Public Key Protocols. IEEE TIT, IT-29(12):198-208, 1983.

[8] H. Gao, P. Degano, C. Bodei and H.R. Nielson. Detecting Replay Attacks by Freshness Annotations. *In Proc. of International Workshop on Issues in the Theory of Security (WITS 2007)*.

[9] H. Gao and H.R. Nielson. Analysis of LySa-calculus with explicit confidentiality annotations. *In Proc. of Advanced Information Networking and Applications (AINA 2006)*, IEEE Computer Society.

[10] A.D. Gordon and A. Jeffrey. Types and Effects for Asymmetric Cryptographic Protocols. *In Proc. of 15th Computer Security Foundations Workshop*, pp. 77-91, IEEE Computer Society, 2002.

[11] J. Heather, G. Lowe and S. Schneider. How to prevent type flaw attacks on security protocols. *In Proc. of the 13th Computer Security Foundations Workshop*, IEEE Computer Society Press, 2000.

[12] Y. Li, W. Yang and J. Huang. *Journal of Information Science and Engineering*, 21:59-84, 2005.

[13] C. Meadows. Identifying potential type confusion in authenticated messages. *In. Proc. of Workshop on Foundation of Computer Security*, pp. 75-84, 2002.

[14] R. Milner. Communicating and mobile systems: the π-calculus. Cambridge University Press, 1999.

[15] H.R. Nielson and F. Nielson. Flow Logic: a multi-paradigmatic approach to static analysis. *The Essence of Computation: Complexity, Analysis, Transformation* LNCS 2566: 223-244, Springer Verlag, 2002.

[16] C.R. Nielsen, F. Nielson, H.R. Nielson. Cryptographic Pattern Matching. ENTCS 168, pp. 91-107, 2007.

[17] M. Satyanarayanan. *Integrating security in a large distributed system.* ACM Transactions on Computer Systems, 7(3):247–280, 1989.

[18] E. Snekkenes. Roles in cryptographic protocols. *In Proc. of the 1992 IEEE Computer Security Symposium on Research in Security and Privacy*, pp.105-119. IEEE Computer Society Press, 1992.

[19] T.Y.C. Woo and S.S. Lam. A lesson on authentication protocol design. *Operating Systems Review*, 28(3):24-37, 1994.

Modelling the Security of Smart Cards by Hard and Soft Types for Higher-Order Mobile Embedded Resources[1]

Mikkel Bundgaard and Thomas Hildebrandt[2]
Jens Chr. Godskesen[3]

IT University of Copenhagen
Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark
{mikkelbu, hilde, jcg}@itu.dk

Abstract

We provide a type system inspired by affine intuitionistic logic for the calculus of Higher-Order Mobile Embedded Resources (Homer), resulting in the first process calculus combining affine linear (non-copyable) and non-linear (copyable) higher-order mobile processes, nested locations, and local names. The type system guarantees that linear resources are neither copied nor embedded in non-linear resources during computation.
We exemplify the use of the calculus by modelling a simplistic e-cash Smart Card system, the security of which depends on the interplay between (linear) mobile hardware, embedded (non-linear) mobile processes, and local names. A purely linear calculus would not be able to express that embedded software processes may be copied. Conversely, a purely non-linear calculus would not be able to express that mobile hardware processes cannot be copied.

Keywords: **Higher-order process passing, linear types, copyable and non-copyable resources, nested locations, security, smart cards**

1 Introduction

Following the seminal work on Mobile Ambients [10], several process calculi, including variations of Mobile Ambients [18,4], the Seal calculus [11], and the Homer calculus [15], have been proposed that combine (1) mobile processes, (2) nested explicit locations and (3) local names. These models are motivated by scenarios in global ubiquitous computing: Mobile processes are employed to represent both mobile *computing devices* (i.e. non-copyable devices such as laptops, PDAs, and smart cards), as well as mobile *computations* (i.e. copyable software agents and

[1] This research is supported by the Danish Research Agency grants no: 274-06-0415 (CosmoBiz), no: 272-05-0258 (Mobile Security) and no: 2059-03-0031 (BPL).

[2] Programming, Logic and Semantics Group.

[3] Computational Logic and Algorithms Group.

migrating processes). Nested explicit locations are typically used to represent administrative domains, firewalls, physical boundaries of mobile devices, boundaries of software messages and processes such as encryption, sand-boxes and locations of data in memory. Finally, local names are used to represent private keys and scope of references to locations in memory.

In the present paper we argue that mobile computing hardware devices are intrinsically *linear*, while mobile computations in software are intrinsically *non-linear*: A hardware device cannot easily be copied, and the security of a system often depends on this fact, for instance for smart cards. Contrarily, software must usually be explicitly protected against copying, e.g. by enclosing it in a tamper-proof (and non-copyable) hardware device.

The Mobile Ambients calculus and its descendants combine linear mobile processes (the ambients) and non-linear, copyable messages (values). These features make the calculi suitable for modelling mobile computing. But since none of the calculi allow general duplication of processes inside ambients, it becomes difficult to represent copyable, mobile computations. On the other hand, the more recent higher-order process calculi, such as the Seal calculus [11] and Homer [15], have explicitly introduced copyable mobile resources in the context of nested locations. But by assuming that *all* resources are copyable these calculi in turn become unrealistic as models of non-copyable mobile computing devices.

Somewhat surprisingly, we found no calculus combining linear and non-linear mobile embedded processes and local names. In the present paper we thus present an extension, inspired by affine intuitionistic linear logic, of the type and effect system for Homer presented in [13]. The extension allows us to distinguish between affine linear and non-linear uses of variables (as in the linear lambda calculus) and to type the names of locations (akin to reference types) and thereby to restrict the content of locations to be either linear or non-linear. We define non-linear to be a subtype of affine linear, which enable non-linear (software) locations to be embedded in linear (hardware) locations. This also ensures that non-linear resources can be used as affine linear resources. On the other hand, the type system guarantees that linear resources are never copied nor embedded in non-linear resources: If a linear resource could be embedded within a non-linear resource it could be copied by copying the embedding resource. In Homer it is possible to reference nested resources using composite location paths. To type these paths we introduce composite reference types guaranteeing that linear locations within non-linear locations are never referenced.

We claim that the calculus captures the intrinsic copyability features of mobile computing hardware devices and software processes as outlined above. We justify this claim by giving in Sec. 5 an example of a simplistic e-cash system, the security of which depends on the non-copyability of smart cards and the ATM itself. Dually, the copyability of software processes, in this case encrypted messages, constitutes an important security threat. We do not claim that the example prove the security of a realistic smart card system, but that it shows that a realistic model for both mobile computing and computations should allow for both linear and non-linear mobile processes.

The type system has consequences for the treatment of infinite behaviours. In

most (untyped) higher-order calculi (HOπ [19], λ-calculus, CHOCS [23], Homer [15]) one can encode infinite behaviour by process passing (and process duplication). Constructors such as the Y-combinator, replication, or general recursion is then taken not as primitives, but rather as derived constructions. However, the encoding of recursion in Homer [15] depends on the ability to copy resources. Thus, we can only encode recursion (and hence replication) of non-linear resources. Since replication does make sense for linear resources, allowing the availability of an arbitrary number of the same resource, we introduce this as a primitive constructor in the calculus.

In the full paper [7], we extend the work in [15] to the linear and non-linearly typed calculus by providing a barbed bisimulation congruence, weak and strong labelled bisimulations, and prove that Howe's method extends to this richer typed setting and thus provides a technique for contextual reasoning about linear and non-linear mobile embedded resources. We show that the labelled bisimulation congruences are sound with respect to the barbed bisimulation congruences, and also complete in the strong case.

The structure of the paper is as follows. In Sec. 2 we present the Homer calculus, and in Sec. 3 we give it transition semantics. The type system for Homer is presented in Sec. 4, and we provide the Smart Card example in Sec. 5. In Sec. 6 we conclude and propose future work.

Related work

The interplay between linearity and non-linearity has been studied thoroughly in variations of Intuitionistic Linear Logic and the corresponding denotational models, term models, and substructural type systems. The models and type systems have been used to describe and reason about co-existing linear and non-linear resources in functional programming, e.g. for memory management and references to system resources [24], in recent languages for quantum computing with classical control [22], and for controlling the use of names (and thus mobility) in the π-calculus [17]. We found no prior studies of linear and non-linear mobile processes combined with nested locations and local names.

The Homer calculus extends Plain CHOCS, but shares ideas with recent calculi for mobility such as the Seal calculus [11], the M-calculus [20] (and its recent successor the Kell calculus [21]), in particular the ability to represent copyable (non-linear), objectively mobile anonymous resources in nested named locations. Type systems have been introduced for the M-calculus (and the Kell calculus [3]) which ensure unity of location names (used for deterministic routing). A type system for Seal calculus is presented in [11], the type system both type active processes and locations, thus enabling one to declare the type of processes that can enter and exit a location.

The composite address paths in Homer are in some respects similar to the composite channel names found in the π-calculus with polyadic synchronisation [9]. In [8] a type system for polyadic synchronisation is given, based on Milner's type system for the polyadic π-calculus. Composite channel names which are typed with the type of the first (or last) element in a composite channel is also suggested, but the idea is not pursued.

Linear types have been studied in great detail in the π-calculus [17,16] by Kobayashi et al. Recently Berger, Honda, and Yoshida [1,2] have investigated the connection between sequential functional computation and typed π-calculus. For a higher-order π-calculus Yoshida and Hennessy [27,26] have examined a type system which captures the effect of mobile processes by typing each process with an interface which describes the resources which the process can access.

A first version of the type system for linear and non-linear resources was proposed in [12] for the calculus of Mobile Resources [14], the predecessor of Homer, but the theory was never developed. Homer was originally presented in [15], together with an adaptation of Howe's method to prove that late contextual bisimulation is a sound characterisation of barbed bisimulation congruence. In [13] the results were extended to prove that input-early strong bisimulation congruence is both a sound and complete characterisation of barbed bisimulation congruence. Homer has also been examined in the setting of bigraphs [5] and its expressivity have been studied in an encoding of the π-calculus [6].

2 Homer

In this section we present the syntax of Homer. The only difference from [13] is that we have extended the syntax with replication.

We assume an infinite set of *names* $\mathcal{N}$ ranged over by m and n, and let $\tilde{n}$ range over finite sets of names. We let δ range over non-empty finite sequences of names, referred to as *paths* and let $\overline{\delta}$ denote *co-paths*. Paths and co-paths are used to reference arbitrarily deeply nested resources. We let φ range over δ and $\overline{\delta}$ and define $\overline{\overline{\delta}} = \delta$. We assume an infinite set of *process variables* $\mathcal{V}$ ranged over by x and y. The sets $\mathbf{p}$ of *process expressions* ranged over by p, $\mathbf{a}$ of *abstractions* ranged over by a, and $\mathbf{c}$ of *concretions* ranged over by c are defined by the grammar:

$$p ::= \mathbf{0} \mid x \mid \varphi e \mid p \parallel p' \mid (n)p \mid !p \ , \qquad e ::= a \mid b \ ,$$
$$a ::= (x)p \ , \qquad b ::= \langle p' : \tilde{n} \rangle p \ , \qquad c ::= b \mid (n)c \ ,$$

where b ranges over unrestricted concretions. We let $\mathbf{t}$, ranged over by t, denote the set $\mathbf{p} \cup \mathbf{a} \cup \mathbf{c}$. Whenever e denotes an abstraction we let $\overline{e}$ denote a concretion, and vice versa. The process p' in the concretion (referred to as the resource) is annotated by a finite set of names. Intuitively, this set of names can be thought of as the names *allocated* by the resource. The annotation is used to control dynamic scope extension when a resource is moved. The annotation is needed because one can define a context that tests if a name is free in a mobile resource. Without the annotations any two processes that do not contain the same names during their computation would be distinguishable (for a full description of this problem and its solution see [13,15]). The type system presented in Sec. 4 guarantees that this set contains all names appearing in the process p'.

The process constructors are the standard constructors from concurrent process calculi: the inactive process, $\mathbf{0}$, process variables, x, action prefixing, φe, parallel composition, $\parallel$, restriction of the name n in p, $(n)p$, and replication of p, $!p$. Homer is defined as a simple extension of the higher-order calculus Plain CHOCS [23] to allow

for *active* processes at named locations denoted by prefixes of the form $n\langle p' : \tilde{n}\rangle p$ and a corresponding prefix denoted by $\overline{n}(x)q$ for moving the process at the location named n and substituting it in for the variable x in q. When the active process is moved, the location disappears (as in Seal) and the residual process is activated. The two prefixes complement the usual prefixes for *passive* process passing in CHOCS denoted by $\overline{n}\langle p' : \tilde{n}\rangle p$ and $n(x)q$. By active and passive we mean that the process p' in the prefix $n\langle p' : \tilde{n}\rangle p$ may perform internal reactions as well as interactions with processes outside the location, whereas the process p' in $\overline{n}\langle p' : \tilde{n}\rangle p$, as in CHOCS, can neither react nor interact with other processes. Interactions with embedded resources are obtained by the use of name *paths*, allowing a process to pass another process to (or to move) an arbitrarily deeply nested active embedded resource. For instance, we have the reductions (ignoring the type annotations)

$$n\langle m(x)q\rangle \parallel \overline{nm}\langle p\rangle p' \longrightarrow n\langle q[^{p}/_{x}]\rangle \parallel p' \tag{1}$$

and

$$n\langle m\langle p\rangle p'\rangle \parallel \overline{nm}(x)q \longrightarrow n\langle p'\rangle \parallel q[^{p}/_{x}] \ , \tag{2}$$

where nm is the name path consisting of the name n followed by the name m.

As usual, (x) bind the variable x and (n) bind the name n. We define the notions of free and bound names ($fn(t)$ and $bn(t)$) and variables of t ($fv(t)$ and $bv(t)$) as standard with the sole exception that $fn(\langle p' : \tilde{n}\rangle p) = \tilde{n} \cup fn(p)$, ie. the annotation determines the free names of a resource. We will call a process without free variables *closed*, and let $\mathbf{t}_c$ and $\mathbf{p}_c$ denote the classes of closed terms and processes, respectively. We will throughout the paper consider terms up to α-equivalence, and we will write $\mathbf{t}_{/\alpha}$ and $\mathbf{p}_{/\alpha}$ for the set of α-equivalence classes of terms and processes, respectively. We will also extend this notion to the sets of closed processes and terms. We use standard shorthands and often elide $\mathbf{0}$ in a process, e.g. writing $\langle p : \tilde{n}\rangle$ instead of $\langle p : \tilde{n}\rangle\mathbf{0}$. For a set of names $\tilde{n} = \{n_1, \ldots, n_k\}$ we will write $(\tilde{n})t$ for $(n_1)\cdots(n_k)t$. We will also write n for the singleton set $\{n\}$ and when convenient let δ and $\overline{\delta}$ denote the set of names in the path. For any two sets s and s' we will write ss' for the union of s and s' under the assumption that $s \cap s' = \emptyset$.

We will say that a relation R on processes is a *congruence* if $\mathbf{0}\ R\ \mathbf{0}$ and $x\ R\ x$, and $p\ R\ p'$ and $q\ R\ q'$ implies $p \parallel q\ R\ p' \parallel q'$, $(n)p\ R\ (n)p'$, $\varphi(x)p\ R\ \varphi(x)p'$, and $\varphi\langle q : \tilde{n}\rangle p\ R\ \varphi\langle q' : \tilde{n}\rangle p'$. We then define *structural congruence* $\equiv$ as the least relation on $\mathbf{p}_{/\alpha}$ that is a congruence and that satisfies the (usual) monoid rules for $(\parallel, \mathbf{0})$ and scope extension as for the π-calculus.

We define the application between an abstraction and a concretion as usual, except that the substitution updates the type annotation in locations.

Definition 2.1 (application and substitution) *For a concretion $c = (\tilde{m})\langle p : \tilde{n}\rangle p'$ and an abstraction $a = (x)p''$, where $\tilde{m} \cap fn(p'') = \emptyset$, we define their application by*

$$c \cdot a = (\tilde{m})(p' \parallel p''[^{p:\tilde{n}}/_{x}]) \quad \textit{and} \quad a \cdot c = (\tilde{m})(p''[^{p:\tilde{n}}/_{x}] \parallel p') \ ,$$

where the capture free substitution $p''[^{p:\tilde{n}}/_{x}]$ is defined inductively in the structure of p'' as usual, except that in the case for concretions, the annotation of the sub-resource is updated, if the variable appears free in the sub-resource. That is, if

Table 1
Transition rules.

$$(\mathit{prefix})\ \frac{}{\varphi e \xrightarrow{\varphi} e} \qquad (\mathit{nesting})\ \frac{p \xrightarrow{\pi} t}{\delta\langle p : \tilde{n}\rangle p' \xrightarrow{\delta\cdot\pi} \delta\langle t : \tilde{n}\rangle p'}$$

$$(\mathit{rest})\ \frac{p \xrightarrow{\pi} t}{(n)p \xrightarrow{\pi} (n)t},\ n \notin \mathit{fn}(\pi) \qquad (\mathit{sync})\ \frac{p \xrightarrow{\varphi} e \quad p' \xrightarrow{\overline{\varphi}} \bar{e}}{p \parallel p' \xrightarrow{\tau} e \cdot \bar{e}}$$

$$(\mathit{par})\ \frac{p \xrightarrow{\pi} t}{p \parallel p' \xrightarrow{\pi} t \parallel p'} \qquad (\mathit{par'})\ \frac{p' \xrightarrow{\pi} t}{p \parallel p' \xrightarrow{\pi} p \parallel t}$$

$$(\mathit{repl1})\ \frac{p \xrightarrow{\pi} t}{!p \xrightarrow{\pi} t \parallel !p} \qquad (\mathit{repl2})\ \frac{p \xrightarrow{\varphi} a \qquad p \xrightarrow{\overline{\varphi}} c}{!p \xrightarrow{\tau} (a \cdot c) \parallel !p}$$

$x \in \mathit{fv}(q)$ *then* $(\langle q : \tilde{m}'\rangle q')[^{p:\tilde{n}}/_x] = \langle q[^{p:\tilde{n}}/_x] : \tilde{m}' \cup \tilde{n}\rangle q'[^{p:\tilde{n}}/_x]$. *If* $x \notin \mathit{fv}(q)$ *then* $(\langle q : \tilde{m}'\rangle q')[^{p:\tilde{n}}/_x] = \langle q : \tilde{m}'\rangle q'[^{p:\tilde{n}}/_x]$.

Note that the substitution discards the type when a variable is reached (see Appendix A for the full definition of application and substitution).

3 Transition semantics

In this section we provide Homer with a labelled transition semantics. As in the previous section, the only difference from [13] is that we have extended the semantics with rules for replication.

We let π range over the set Π of labels, defined as $\pi ::= \tau \mid \varphi$ (recall $\varphi ::= \delta \mid \overline{\delta}$). The set of free names in π, $\mathit{fn}(\pi)$, is $\mathit{fn}(\delta)$ whenever $\pi = \delta$ or $\pi = \overline{\delta}$ and $\emptyset$ otherwise. The rules in Table 1 then define a labelled transition system

$$(\mathbf{t}_{c/\alpha}, \longrightarrow \subseteq \mathbf{p}_{c/\alpha} \times \Pi \times \mathbf{t}_{c/\alpha})$$

for α-equivalence classes of closed processes.

To allow for a more succinct presentation of the transitions of nested active resources we close concretions and abstractions under process operators. Hence, whenever $c = (\tilde{n})\langle p_1 : \tilde{n}_1\rangle p$ and assuming $\tilde{n} \cap (\mathit{fn}(p') \cup n \cup \delta) = \emptyset$ (using α-conversion if needed) we let $c \parallel p'$ denote $(\tilde{n})\langle p_1 : \tilde{n}_1\rangle(p \parallel p')$, we let $(n)c$ denote $(n\tilde{n})\langle p_1 : \tilde{n}_1\rangle p$, if $n \in \tilde{n}_1$ and otherwise it denotes $(\tilde{n})\langle p_1 : \tilde{n}_1\rangle(n)p$, and we let $\delta\langle c : \tilde{n}'\rangle p'$ denote $(\tilde{n})\langle p_1 : \tilde{n}_1\rangle\delta\langle p : \tilde{n}'\tilde{n}\rangle p'$. Similarly, whenever $a = (x)p$ and assuming $x \notin \mathit{fv}(p')$ (using α-conversion if needed) we let $a \parallel p'$ denote $(x)(p \parallel p')$, $(n)a$ for $(x)(n)p$, and we let $\delta\langle a : \tilde{n}\rangle p'$ denote $(x)\delta\langle p : \tilde{n}\rangle p'$. These shorthands are applied in the rules (*nesting*), (*rest*), (*par*), (*par'*) and (*repl1*).

The rules conservatively extend the rules for Plain CHOCS. Note that the rule (*sync*) covers the two different kind of interactions: the active and passive resource movement as described in the previous section, and that the rule (*nesting*) permits arbitrarily deeply nested active resources to be moved, receive resources, and perform internal computation steps. To allow these three kinds of actions we use an

operation $\delta \cdot (\_)$ for extending location paths, defined by:

$$\delta \cdot \tau = \tau, \quad \delta \cdot \delta' = \delta\delta' \ .$$

Note that the operation is not defined for $\overline{\delta}$ since $\overline{\delta}$ is directed "downward" and thus not visible outside the resource. Since $\delta \cdot \tau = \tau$, the nesting rule implies that $\delta\langle p : \tilde{n}\rangle p' \xrightarrow{\tau} \delta\langle t : \tilde{n}\rangle p'$, if $p \xrightarrow{\tau} t$.

As an example of using the rules (and shorthands for concretions and abstractions) the reduction (1) in the previous section can be derived from $m(x)q \xrightarrow{m} (x)q$, so $n\langle m(x)q : \tilde{n}\rangle \xrightarrow{nm} (x)n\langle q : \tilde{n}\rangle$. Combining with $\overline{nm}\langle p : \tilde{n}'\rangle p' \xrightarrow{\overline{nm}} \langle p : \tilde{n}'\rangle p'$ we obtain

$$n\langle m(x)q : \tilde{n}\rangle \parallel \overline{nm}\langle p : \tilde{n}'\rangle p' \xrightarrow{\tau} (x)n\langle q : \tilde{n}\rangle \cdot \langle p : \tilde{n}'\rangle p' \ .$$

By Def. 2.1 we get $(x)n\langle q : \tilde{n}\rangle \cdot \langle p : \tilde{n}'\rangle p' = n\langle q[^{p:\tilde{n}'}/_x] : \tilde{n} \cup \tilde{n}'\rangle \parallel p'$ (if $x \in fv(q)$).

Similarly for the reduction (2) we have that $m\langle p : \tilde{n}'\rangle p' \xrightarrow{m} \langle p : \tilde{n}'\rangle p'$, so

$$n\langle m\langle p : \tilde{n}'\rangle p' : \tilde{n}''\rangle \xrightarrow{nm} \langle p : \tilde{n}'\rangle n\langle p' : \tilde{n}''\rangle \ .$$

Combining this transition with $\overline{nm}(x)q \xrightarrow{\overline{nm}} (x)q$ we obtain

$$n\langle m\langle p : \tilde{n}'\rangle p' : \tilde{n}''\rangle \parallel \overline{nm}(x)q \xrightarrow{\tau} \langle p : \tilde{n}'\rangle n\langle p' : \tilde{n}''\rangle \cdot (x)q \ ,$$

which by Def. 2.1 is the process $n\langle p' : \tilde{n}''\rangle \parallel q[^{p:\tilde{n}'}/_x]$.

4 Type system

We are now ready to present the extension of the type and effect system given for Homer in [13] to allow a distinction between affine linear and non-linear resources.

We will assume a set $\mathcal{S} = \{\texttt{aff}, \texttt{un}\}$, of *affine* and *unrestricted* (i.e. non-linear) *sorts*, and let S range over sorts. Furthermore, we will assume the subtyping relation $\leq$ on $\mathcal{S}$ such that $\texttt{un} < \texttt{aff}$, which corresponds with our intuition that an unrestricted process can be used instead of an affine process. Or concretely, as exemplified by the model of the e-cash system in Sec. 5 below, that software can be embedded in, and used as, hardware, but not the other way around.

Process types consist of two parts written as $S\ \tilde{n}$. The first part, the sort S, records if the process is affine linear or non-linear. The second part, $\tilde{n}$, was introduced by the type system in [13] and can be regarded as an effect that captures the names used or allocated by the process, as described in Sec. 2. The type system guarantees that this set is a superset of the free names in the process. Besides process types, we also define concretion and abstraction types. The *concretion type* $\langle S\rangle S'\ \tilde{n}'$ types a concretion $(\tilde{m}')\langle p : \tilde{m}\rangle p'$ in which the transferred process p has sort S and where the entire concretion has the sort S' and effect $\tilde{n}'$. The *abstraction type* $S \twoheadrightarrow S'\ \tilde{n}$ types an abstraction $(x)p$ that itself has sort S' and effect $\tilde{n}$ and accepts a process of sort S. We will only consider abstraction and concretion types where $S \leq S'$, and this is ensured by the typing rules.

Table 2
Typing address paths.

$$\frac{}{\Gamma, n : S \vdash n : S \texttt{ Ref } S} \qquad \frac{\Gamma, n : S \vdash \delta : S'' \texttt{ Ref } S'}{\Gamma, n : S \vdash \delta n : S'' \texttt{ Ref } S}(S \leq S') \qquad \frac{\Gamma \vdash \delta : S \texttt{ Ref } S'}{\Gamma \vdash \overline{\delta} : S \texttt{ Ref } S'}$$

Definition 4.1 (types) *We define three kinds of types,* process types T_p, concretion types T_c, *and* abstraction types T_a, *by the following grammar*

$$T ::= T_p \mid T_c \mid T_a$$
$$T_p ::= S\ \tilde{n}\ , \quad T_c ::= \langle S \rangle T_p\ , \quad T_a ::= S \twoheadrightarrow T_p$$

For $n \notin \tilde{n}$ we write $(S\ \tilde{n})n$ for the process type $S\ \tilde{n}n$ and $(\langle S \rangle S'\ \tilde{n})n$ for the concretion type $\langle S \rangle S'\ \tilde{n}n$. We write $T \cup \tilde{n}''$ for the (not necessarily disjoint) name extension of the type T defined by

$$\begin{aligned}(S\ \tilde{n}) \cup \tilde{n}'' &= S\ \tilde{n} \cup \tilde{n}'' \\ (\langle S \rangle T_p) \cup \tilde{n}'' &= \langle S \rangle T_p \cup \tilde{n}'' \\ (S \twoheadrightarrow T_p) \cup \tilde{n}'' &= S \twoheadrightarrow T_p \cup \tilde{n}''\ .\end{aligned}$$

Type environments Γ assign sorts to names and variables.

Definition 4.2 (type environment) *A type environment* Γ *is a partial function* $\Gamma : \mathcal{N} \uplus \mathcal{V} \rightharpoonup \mathcal{S}$ *from names and variables to sorts. We will write* $dom_n(\Gamma)$ *and* $dom_v(\Gamma)$ *for respectively names and variables in the domain of* Γ, *and let* $dom(\Gamma) = dom_n(\Gamma) \cup dom_v(\Gamma)$. *If* $n \notin dom_n(\Gamma)$ *we write* $\Gamma, n : S$ *for the extension of* Γ *with the mapping from* n *to* S, *and similarly for variables. We will let* Δ *range over environments with no variable mappings.*

To present our typing rules we need to be able to combine two environments in a way that, as usual for linear type systems, constrain the presence of linearly used variables. Letting l range over both names and variables, we define the combination Γ'' of two type environments Γ and Γ', denoted $\Gamma \odot \Gamma' = \Gamma''$, by $\Gamma \cup \Gamma'$ if $\{x \mid \Gamma(x) = \texttt{aff}\} \cap \{x' \mid \Gamma'(x') = \texttt{aff}\} = \emptyset$, and if $l \in dom(\Gamma) \cap dom(\Gamma')$ implies $\Gamma(l) = \Gamma'(l)$. The requirements enforce that for $\Gamma \odot \Gamma' = \Gamma''$, any name occurring in Γ'' can either occur in Γ, Γ', or in both (if it has the same sort). The same is the case for unrestricted variables, whereas the same *affine linear* variable cannot in occur in both Γ and Γ'. This underlines, that our type system is concerned with linear use of *processes* and not of names, as in [17].

We also need typing of *address paths*: $\Gamma \vdash \varphi : S \texttt{ Ref } S'$, as defined by the rules in Table 2. The type $S \texttt{ Ref } S'$ is read as *a reference via* S *to* S'. The rules ensure that the sorts of the names in an address path typed $S \texttt{ Ref } S'$ form a non-strictly descending chain, ensuring that an affine resource cannot be referenced inside an unrestricted resource, and that the first name of the address path has sort S and the last name of the path has sort S'. For instance, letting $\Gamma = m : \texttt{aff}, n : \texttt{un}$, we can derive $\Gamma \vdash mm : \texttt{aff Ref aff}$ and $\Gamma \vdash mmn : \texttt{aff Ref un}$, but we cannot derive neither $\Gamma \vdash nm : \texttt{un Ref aff}$ nor $\Gamma \vdash mnm : \texttt{aff Ref aff}$.

Table 3
Typing rules for affine linear and non-linear Homer

$$(\textit{variable})\ \frac{}{\Gamma, x : S \vdash x : S\ \tilde{n}}(\tilde{n} \subseteq dom_n(\Gamma)) \qquad (\textit{inactive})\ \frac{}{\Gamma \vdash \mathbf{0} : \mathtt{un}\ \tilde{n}}(\tilde{n} \subseteq dom_n(\Gamma))$$

$$(\textit{parallel})\ \frac{\Gamma \vdash p : S\ \tilde{n} \quad \Gamma' \vdash p' : S\ \tilde{n}'}{\Gamma \odot \Gamma' \vdash p \parallel p' : S\ \tilde{n} \cup \tilde{n}'} \qquad (\textit{rest})\ \frac{\Gamma, n : S \vdash p : T_p n}{\Gamma \vdash (n)p : T_p}$$

$$(\textit{rest-conc})\ \frac{\Gamma, n : S \vdash (\tilde{m})\langle p : \tilde{m}n\tilde{n}\rangle p' : T_c n}{\Gamma \vdash (\tilde{m}n)\langle p : \tilde{m}n\tilde{n}\rangle p' : T_c} \qquad (\textit{embed})\ \frac{\Gamma \vdash p : \mathtt{un}\ \tilde{n}}{\Gamma \vdash p : \mathtt{aff}\ \tilde{n}}$$

$$(\textit{repl})\ \frac{\Gamma \vdash p : T_p}{\Gamma \vdash\ !p : T_p}(\forall x \in fv(p).\ \Gamma(x) = \mathtt{un}) \qquad (\textit{abs})\ \frac{\Gamma, x : S' \vdash p : S\ \tilde{n}}{\Gamma \vdash (x)p : S' \twoheadrightarrow S\ \tilde{n}}(S' \leq S)$$

$$(\textit{conc})\ \frac{\Gamma \vdash p : S\ \tilde{n} \quad \Gamma' \vdash p' : S'\ \tilde{n}'}{\Gamma \odot \Gamma' \vdash \langle p : \tilde{n}\rangle p' : \langle S\rangle S'\ \tilde{n} \cup \tilde{n}'}(S \leq S')$$

$$(\textit{pre-abs})\ \frac{\Gamma \vdash a : S' \twoheadrightarrow S\ \tilde{n} \quad \Gamma \vdash \varphi : S''\ \mathtt{Ref}\ S'}{\Gamma \vdash \varphi a : S\ \tilde{n} \cup \varphi}(S'' \leq S)$$

$$(\textit{pre-conc})\ \frac{\Gamma \vdash b : \langle S'\rangle S\ \tilde{n} \quad \Gamma \vdash \varphi : S''\ \mathtt{Ref}\ S'}{\Gamma \vdash \varphi b : S\ \tilde{n} \cup \varphi}(S'' \leq S)$$

We define the typing of processes, abstractions, and concretions using the rules in Table 3. The type system conservatively generalises the prior type (effect) system for Homer [13], which we can obtain by removing the (*embed*) rule and taking $\mathcal{S}$ to be a singleton set, making it possible to delete all references to sorts from abstraction and concretion types, and completely remove side-conditions and environments. We only explain some of the rules, the rest should be self-explanatory. The (*conc*) rule allows us to type a basic concretion, if the extruded process has a sub-sort of the residual process. We can type an abstraction with (*abs*), if we can type the body of the abstraction under an extended environment, where x is given a sub-sort of the sort of the abstraction. The rule (*pre-abs*) allows us to form a process from an abstraction as long as the sort of the received process is the sort that the abstraction expects from the address path. The rule (*embed*) corresponds to the usual subsumption rule in type systems with subtyping, concretely it allows us to treat unrestricted processes as affine processes. The side-condition in the rule (*repl*) ensures us that all variables in Γ that occur free in p are unrestricted, however Γ may contain affine variables which do not occur free in p.

The typing rules for processes employ the path types to make sure that the resource provider and receiver agrees on what is being communicated, combining ideas of reference types, which constrain the types of the referenced resources, and types for process calculi, which constrain the types of objects being communicated on channels. Thus for a typed address path $\Gamma \vdash \varphi : S\ \mathtt{Ref}\ S'$ both the resource provider and receiver agree on that the communicated process has sort S' (this constraint can be weakened by subsumption for the provider's part, and narrowing for the receiver's part). The sort S of the outermost name of the address path in

the path type is used in the side-conditions of the rules (*pre-conc*) and (*pre-abs*) to ensure that any process using a path has a super-sort of S, which means that affine names can never occur in paths inside unrestricted resources. For instance, if n is affine and m is unrestricted then in the process $nm\langle p : \tilde{n}'\rangle q : S\ \tilde{n}$ the resource p is unrestricted, but the typing rules enforce that $S = \mathtt{aff}$, meaning that the entire process is typed as affine. This is a restricted use of linear resources, but it fits well with the scenario of linear, mobile computing devices containing non-linear mobile computations: A mobile computing device can never be contained in or manipulated by a software process.

We have implemented a typing algorithm by eliminating the rule (*embed*) and following the approach for linear type systems [25]. The typing algorithm requires that we annotate name restriction with a sort, as we cannot infer the correct sort from the restriction. See the full paper for this algorithm [7].

We can prove the standard properties about the type system: strengthening of unused names and variables, invariance under structural congruence etc. Again, we refer to the full paper for these results [7] and only present the main results here. As expected in a type system with subtyping we have narrowing of variables.

Proposition 4.3 (narrowing of variables) *If $\Gamma, x : S \vdash t : T$ and $S' \leq S$ then $\Gamma, x : S' \vdash t : T$.*

Note that we in general cannot use narrowing (or widening) for names, as this can make address paths ill-typed, i.e. the ordering can be destroyed, if we allow to change the type of a name.

Lemma 4.4 (substitution lemma) *Let $\Delta \vdash p : S\ \tilde{n}$ be a closed process and let $\Gamma', x : S \vdash t' : T'$ be a term with $\Delta \odot \Gamma'$ defined then $\Delta \odot \Gamma' \vdash t'[{}^{p:\tilde{n}}/_{x}] : T''$, where $T'' = T' \cup \tilde{n}$ if $x \in fv(t')$ and $T'' = T'$ otherwise.*

Our type system ensures us that well-typed terms satisfies several properties, below we state the main properties. The properties imply that the annotation of resources contains the free names of the resource, that affine terms cannot be contained in unrestricted terms, and that affine terms cannot be duplicated.

Lemma 4.5 (properties of well-typed terms) *Writing $n(T)$ for the names and $s(T)$ for the sort of the type T, defined as $\tilde{n}$ and S, if T is of the form $S\ \tilde{n}$, $S' \twoheadrightarrow S\ \tilde{n}$, or $\langle S'\rangle S\ \tilde{n}$. If $\Gamma \vdash t : T$ then*

- *$fn(t) \subseteq n(T) \subseteq dom_n(\Gamma)$ and $fv(t) \subseteq dom_v(\Gamma)$.*
- *If $x : \mathtt{aff} \in \Gamma$ then x occurs free at most once in t.*
- *If $x : \mathtt{aff} \in \Gamma$ and $x \in fv(t)$ then $s(T) = \mathtt{aff}$.*
- *If $s(T) = \mathtt{un}$ then for every sub-derivation $\Gamma' \vdash t' : T'$ we have $s(T') = \mathtt{un}$.*

Theorem 4.6 (subject reduction, labelled transition relation) *Suppose $\Gamma \vdash p : S\ \tilde{n}$ and $p \xrightarrow{\pi} t$ then one of the following cases hold.*

- *$\pi = \tau$, $t = p'$ and $\Gamma \vdash p' : S\ \tilde{n}$.*
- *$\pi = \varphi$, $t = a$ and $\Gamma \vdash a : S' \twoheadrightarrow S\ \tilde{n}$ and $\Gamma \vdash \varphi : S''\ \mathtt{Ref}\ S'$ for some S' and $S'' \leq S$.*
- *$\pi = \varphi$, $t = c$ and $\Gamma \vdash c : \langle S'\rangle S\ \tilde{n}$ and $\Gamma \vdash \varphi : S''\ \mathtt{Ref}\ S'$ for some S' and $S'' \leq S$.*

5 An e-cash Smart Card application

In this section we provide a simple model of an e-cash system that illustrates the combination of linear and non-linear mobile resources, nested locations, and local names. Consider first a process defined by

$$crypt_{e,k} = e(x)e\langle k\langle x : \emptyset\rangle : \{k\}\rangle \ .$$

The process is able to receive a resource on the name e, which is then placed inside a location named k nested in a location named e. If k is cryptographic key, one can think of the process as being able to perform a single encryption of a process (or message) communicated on the public channel e. This can be utilised in a simple e-cash system consisting of an ATM that is able to provide a coin $\overline{c}\langle \mathbf{0} : \emptyset\rangle$, if the process in the location v can encrypt a nonce n with the private key k:

$$\begin{aligned} atm &= (k)\big(v\langle crypt_{e,k} : \{e,k\}\rangle \parallel cash_k\big) \\ cash_k &= !(n)\left(\overline{ve}\langle n\langle \mathbf{0} : \emptyset\rangle : \{n\}\rangle\overline{vekn}(x)\overline{c}\langle \mathbf{0} : \emptyset\rangle\right) \ . \end{aligned}$$

In the control process $cash_k$ of the ATM a nonce process $n\langle \mathbf{0} : \emptyset\rangle$ is sent to the location e inside the process in the location v. Subsequently, a process is retrieved from the sub location $vekn$. If this succeeds, it must be the case that the process inside the location v has embedded the nonce in the location k, and the ATM then emits a coin. Hence we get the following sequence of transitions

$$\begin{aligned} atm \xrightarrow{\tau} &\equiv (k)\big((n')(v\langle e\langle k\langle n'\langle \mathbf{0} : \emptyset\rangle : \{n'\}\rangle : \{k,n'\}\rangle : \{e,k,n'\}\rangle \parallel \\ &\qquad \overline{vekn'}(x)\overline{c}\langle \mathbf{0} : \emptyset\rangle) \parallel cash_k\big) \\ \xrightarrow{\tau} &\equiv (k)\,(p \parallel cash_k) \parallel \overline{c}\langle \mathbf{0} : \emptyset\rangle \ , \end{aligned}$$

where $p =_{def} (n')\,(v\langle e\langle k\langle \mathbf{0} : \{n'\}\rangle : \{k,n'\}\rangle : \{e,k,n'\}\rangle)$ is a slot containing a "used" smart card, i.e. where the nonce has been removed.

The control process can potentially be executed any number of times. The intended behaviour is however, that only one coin will ever be delivered, since the method on the card can only encrypt once. Alas, if the process in the slot v can be copied, the security is broken. A e-cash copying thief may be defined by

$$thief = \overline{v}(x)\,(v\langle x : \emptyset\rangle \parallel v\langle x : \emptyset\rangle) \ ,$$

which picks up the e-cash process by $\overline{v}(x)$ and creates two copies. Then (again letting $p =_{def} (n')\,(v\langle e\langle k\langle \mathbf{0} : \{n'\}\rangle : \{k,n'\}\rangle : \{e,k,n'\}\rangle)$) security will break down

$$\begin{aligned} atm \parallel thief \xrightarrow{\tau} &\equiv \quad (k)\big(v\langle crypt_{e,k} : \{e,k\}\rangle \parallel v\langle crypt_{e,k} : \{e,k\}\rangle \parallel cash_k\big) \\ \xrightarrow{\tau}{}^{*} &\equiv \quad (k)\,(p \parallel p \parallel cash_k) \parallel \overline{c}\langle \mathbf{0} : \emptyset\rangle \parallel \overline{c}\langle \mathbf{0} : \emptyset\rangle \ . \end{aligned}$$

The type system presented in the previous section allows us to type the location v as affine linear. Thereby, we can model that the process in location v is intended

as being embedded in a non-copyable smart card (and also ensure that the entire system cannot be copied either). First, we show that the system is well-typed.

Lemma 5.1 *Let* $\Delta = e : \mathtt{un}, c : \mathtt{aff}, v : \mathtt{aff}$*, then* $\Delta \vdash atm : \mathtt{aff}\ \{e, c, v\}$.

We then show, that we cannot type the system $atm \parallel thief$, if the slot v is linear, as this makes it impossible to copy the content of the slot, i.e. the smart card.

Proposition 5.2 *For any* $\Delta, v : \mathtt{aff}$*,* $\tilde{n}$ *and sort* S *it is not possible to derive* $\Delta, v : \mathtt{aff} \vdash atm \parallel thief : S\ \tilde{n}$.

Proof (Sketch) Assume that it is possible to derive $\Delta, v : \mathtt{aff} \vdash atm \parallel thief : S\ \tilde{n}$, by inspecting the derivation, and without loss of generality, is must also be possible to derive $\Delta, v : \mathtt{aff}, x : \mathtt{aff} \vdash v\langle x : \emptyset\rangle \parallel v\langle x : \emptyset\rangle : S\ \tilde{n}$, but this contradicts Lemma 4.5 (that x occurs free at most once). □

Note that the encrypted nonce is unrestricted. The security would be broken, if we repeatedly had used the same secret name n as challenge for the card, i.e. swapping the local name (n) and the replication in the definition of the control process, defining $cash_k$ as $(n)!\left(\overline{ve}\langle n\langle \mathbf{0} : \emptyset\rangle : \{n\}\rangle\overline{vekn}(x)\overline{c}\langle \mathbf{0} : \emptyset\rangle\right)$, A thief which interrupts the ATM just after the name n has been send (and encrypted at the card) and which copies the encrypted content of the card could be defined by

$$thief' = \overline{ve}(x)(v\langle e\langle x : \emptyset\rangle : \{e\}\rangle \parallel ve(x')v\langle e\langle x : \emptyset\rangle : \{e\}\rangle)\ ,$$

where the right-hand side of the parallel composition receives and discards the challenge message the second time it is send by the ATM, and provides a card with the copied encrypted content. Letting

$$\begin{aligned} p &=_{def} !\left(\overline{ve}\langle n\langle \mathbf{0} : \emptyset\rangle : \{n\}\rangle\overline{vekn}(x)\overline{c}\langle \mathbf{0} : \emptyset\rangle\right), \\ q &=_{def} v\langle e\langle k\langle \mathbf{0} : \{n\}\rangle : \{k, n\}\rangle : \{e, k, n\}\rangle, \text{ and} \\ q' &=_{def} v\langle e\langle k\langle n\langle \mathbf{0} : \emptyset\rangle : \{n\}\rangle : \{k, n\}\rangle : \{e, k, n\}\rangle \end{aligned}$$

we have the following transitions

$$\begin{aligned} atm \parallel thief' &\xrightarrow{\tau}{}^{*} \equiv \{k, n\}\left(v\langle \mathbf{0} : \{e, k, n\}\rangle \parallel \overline{vekn}(x)\overline{c}\langle \mathbf{0} : \emptyset\rangle \parallel p \parallel q' \parallel ve(x')q'\right) \\ &\xrightarrow{\tau}{}^{*} \equiv \{k, n\}\left(v\langle \mathbf{0} : \{e, k, n\}\rangle \parallel p \parallel q \parallel q\right) \parallel \overline{c}\langle \mathbf{0} : \emptyset\rangle \parallel \overline{c}\langle \mathbf{0} : \emptyset\rangle\ . \end{aligned}$$

This security threat would not show in a purely linear calculus. We leave for future work to apply the bisimulation congruence presented in [7] to prove that the typed *atm* is indeed secure in any context.

6 Conclusions and future work

We have successfully extended the prior type and effect system for Homer to provide the first process calculus combining affine linear and non-linear nested mobile embedded processes with local names. By a concrete e-cash Smart Card system we

have exemplified that the calculus captures the difference between mobile *computing* hardware and embedded mobile software *computations*, which is crucial for the security of pervasive and ubiquitous computing.

We believe that the type system presented for Homer in the present paper can be adapted to other calculi combining mobile embedded resources with local names, as for instance Mobile Ambients and the Seal calculus. We expect to investigate other variations and applications of linear types and more expressive type systems for Homer within the research projects for Mobile Security and Computer Supported Mobile Adaptive Business Processes (CosmoBiz) at the IT-University of Copenhagen.

References

[1] Berger, M., K. Honda and N. Yoshida, *Sequentiality and the π-calculus*, in: S. Abramsky, editor, *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications (TLCA'01)*, Lecture Notes in Computer Science **2044** (2001), pp. 29–45.

[2] Berger, M., K. Honda and N. Yoshida, *Genericity and the π-calculus*, Acta Informatica **42** (2005), pp. 83–141.

[3] Bidinger, P. and J.-B. Stefani, *The Kell calculus: Operational semantics and type system*, in: E. Najm, U. Nestmann and P. Stevens, editors, *Proceedings of the 5th IFIP International Conference on Formal Methods for Object-Based Distributed Systems (FMOODS'03)*, Lecture Notes in Computer Science **2884** (2003), pp. 109–123.

[4] Bugliesi, M., G. Castagna and S. Crafa, *Access control for mobile agents: The calculus of boxed ambients*, ACM Transactions on Programming Languages and Systems (TOPLAS) **26** (2004), pp. 57–124.

[5] Bundgaard, M. and T. Hildebrandt, *Bigraphical semantics of higher-order mobile embedded resources with local names*, in: A. Rensink, R. Heckel and B. König, editors, *Proceedings of the Graph Transformation for Verification and Concurrency workshop (GT-VC'05)*, Electronic Notes in Theoretical Computer Science **154** (2006), pp. 7–29.

[6] Bundgaard, M., T. Hildebrandt and J. C. Godskesen, *A CPS encoding of name-passing in higher-order mobile embedded resources*, Theoretical Computer Science **356** (2006), pp. 422–439.

[7] Bundgaard, M., T. Hildebrandt and J. C. Godskesen, *Typing linear and non-linear higher-order mobile embedded resources with local names*, Technical Report TR-2007-97, IT University of Copenhagen (2007), available from http://www.itu.dk/~mikkelbu/typedHomer.pdf.

[8] Carbone, M., "Trust and Mobility," Ph.D. thesis, BRICS (2005).

[9] Carbone, M. and S. Maffeis, *On the expressive power of polyadic synchronisation in π-calculus*, Nordic Journal of Computing **10** (2003), pp. 70–98.

[10] Cardelli, L. and A. D. Gordon, *Mobile ambients*, Theoretical Computer Science **240** (2000), pp. 177–213.

[11] Castagna, G., J. Vitek and F. Z. Nardelli, *The Seal calculus*, Journal of Information and Computation **201** (2005), pp. 1–54.

[12] Godskesen, J. C. and T. Hildebrandt, *Copyability types for mobile computing resources* (2004), presented at the International Workshop on Formal Methods and Security, Nanjing, China.

[13] Godskesen, J. C. and T. Hildebrandt, *Extending Howe's method to early bisimulations for typed mobile embedded resources with local names*, in: *Proceedings of the 25th Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS'05)*, Lecture Notes in Computer Science **3821** (2005), pp. 140–151.

[14] Godskesen, J. C., T. Hildebrandt and V. Sassone, *A calculus of mobile resources*, in: L. Brim, P. Jancar, M. Kretínský and A. Kucera, editors, *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR'02)*, Lecture Notes in Computer Science **2421** (2002), pp. 272–287.

[15] Hildebrandt, T., J. C. Godskesen and M. Bundgaard, *Bisimulation congruences for Homer — a calculus of higher order mobile embedded resources*, Technical Report TR-2004-52, IT University of Copenhagen (2004).

[16] Kobayashi, N., *Type systems for concurrent programs* (2002), in *Proceedings of UNU/IIST 10th Anniversary Colloquium*.

[17] Kobayashi, N., B. C. Pierce and D. N. Turner, *Linearity and the pi-calculus*, ACM Transactions on Programming Languages and Systems (TOPLAS) **21** (1999), pp. 914–947.

[18] Levi, F. and D. Sangiorgi, *Mobile safe ambients*, ACM Transactions on Programming Languages and Systems (TOPLAS) **25** (2003), pp. 1–69.

[19] Sangiorgi, D., "Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms," Ph.D. thesis, Department of Computer Science, University of Edinburgh (1992).

[20] Schmitt, A. and J.-B. Stefani, *The M-calculus: A higher-order distributed process calculus*, in: *Proceedings of the 30th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'03)* (2003), pp. 50–61.

[21] Schmitt, A. and J.-B. Stefani, *The Kell calculus: A family of higher-order distributed process calculi*, in: C. Priami and P. Quaglia, editors, *Proceedings of the International Workshop on Global Computing Workshop (GC'04)*, Lecture Notes in Computer Science **3267** (2004), pp. 146–178.

[22] Selinger, P. and B. Valiron, *A lambda calculus for quantum computation with classical control*, Journal of Mathematical Structures in Computer Science **16** (2006), pp. 527–552.

[23] Thomsen, B., *Plain CHOCS: A second generation calculus for higher order processes*, Acta Informatica **30** (1993), pp. 1–59.

[24] Turner, D. N. and P. Wadler, *Operational interpretations of linear logic*, Theoretical Computer Science **227** (1999), pp. 231–248.

[25] Walker, D., *Substructural type systems*, in: B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, MIT Press, 2004 pp. 3–43.

[26] Yoshida, N., *Channel dependent types for higher-order mobile processes (extended abstract)*, in: N. D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'04)* (2004), pp. 147–160.

[27] Yoshida, N. and M. Hennessy, *Assigning types to processes*, Journal of Information and Computation **174** (2004), pp. 143–179.

A Application and substitution

Definition A.1 (application and substitution) *Given a concretion $c = (\tilde{m})\langle p : \tilde{n}\rangle p'$ and an abstraction $a = (x)p''$ we define their application as follows whenever $\tilde{m} \cap fn(p'') = \emptyset$*

$$c \cdot a = (\tilde{m})(p' \parallel p''[^{p:\tilde{n}}/_x]) \quad \textit{and} \quad a \cdot c = (\tilde{m})(p''[^{p:\tilde{n}}/_x] \parallel p')$$

where $p''[^{p:\tilde{n}}/_x]$ is defined inductively in the structure of p''.

$$\begin{array}{lll}
\mathbf{0}[^{p:\tilde{n}}/_x] & = \mathbf{0} & \\
x[^{p:\tilde{n}}/_x] & = p & \\
x'[^{p:\tilde{n}}/_x] & = x' & \textit{if } x \neq x' \\
(q \parallel q')[^{p:\tilde{n}}/_x] & = q[^{p:\tilde{n}}/_x] \parallel q'[^{p:\tilde{n}}/_x] & \\
((n)q)[^{p:\tilde{n}}/_x] & = (n)(q[^{p:\tilde{n}}/_x]) & \textit{if } n \notin \tilde{n} \\
(!q)[^{p:\tilde{n}}/_x] & = !(q[^{p:\tilde{n}}/_x]) & \\
(\varphi e)[^{p:\tilde{n}}/_x] & = \varphi e[^{p:\tilde{n}}/_x] & \\
(\langle q : \tilde{m}'\rangle q')[^{p:\tilde{n}}/_x] & = \langle q[^{p:\tilde{n}}/_x] : \tilde{m}' \cup \tilde{n}\rangle q'[^{p:\tilde{n}}/_x] & \textit{if } x \in fv(q) \\
(\langle q : \tilde{m}'\rangle q')[^{p:\tilde{n}}/_x] & = \langle q : \tilde{m}'\rangle q'[^{p:\tilde{n}}/_x] & \textit{if } x \notin fv(q) \\
((x')q)[^{p:\tilde{n}}/_x] & = (x')(q[^{p:\tilde{n}}/_x]) & \textit{if } x \neq x'
\end{array}$$

B Results

We will write φ_i for the i'th element of the path φ, and $\mathit{length}(\varphi)$ for the length of the path φ.

Proposition B.1 *$dom_n(\Gamma) \supseteq \varphi$ and $\forall i, j$ with $1 \leq i \leq j \leq \mathit{length}(\varphi)$ we have $\Gamma(\varphi_j) \leq \Gamma(\varphi_i)$ and $\Gamma(\varphi_1) = S$ and $\Gamma(\varphi_{\mathit{length}(\varphi)}) = S'$ iff $\Gamma \vdash \varphi : S$ `Ref` S'.*

We combine both weakening propositions in one, and let l range over names and variables.

Proposition B.2 (weakening) *If $\Gamma \vdash t : T$ and $l \notin dom(\Gamma)$ then $\Gamma, l : S \vdash t : T$.*

Proposition B.3 (strengthening, names) *Assume $n \notin fn(t)$ and $\Gamma, n : S \vdash t : T$ then $\Gamma \vdash t : T \setminus n$.*

Proposition B.4 (strengthening, variables) *Assume $x \notin fv(t)$ and $\Gamma, x : S \vdash t : T$ then $\Gamma \vdash t : T$.*

Proposition B.5 *If $\Gamma \vdash t : T$ and $n : S' \in \Gamma$ then $\Gamma \vdash t : T \cup n$.*

Proposition B.6 (structural congruence and typing) *If $f \equiv f'$ then $\Gamma \vdash t : T$ iff $\Gamma \vdash f' : T$.*

Proposition B.7 (well-typed application) *If $\Gamma \vdash a : S'' \twoheadrightarrow S'\ \tilde{n}''$ is an closed abstraction and $\Gamma' \vdash c : \langle S''\rangle S'\ \tilde{n}'$ is a closed concretion with $c \cdot a$ and $\Gamma \odot \Gamma'$ defined then $\Gamma \odot \Gamma' \vdash c \cdot a : S'\ \tilde{n}'' \cup \tilde{n}'$ is a closed process.*

State-oriented noninterference for CCS

Ilaria Castellani[1,2]

INRIA Sophia Antipolis
2004 route des Lucioles, BP 93,
06902 Sophia Antipolis Cedex, France

Abstract

We address the question of typing *noninterference* (NI) in the calculus CCS, in such a way that Milner's translation into CCS of a standard parallel imperative language preserves both an existing NI property and the associated type system. Recently, Focardi, Rossi and Sabelfeld have shown that a variant of Milner's translation, restricted to the sequential fragment of the language, maps a time-sensitive NI property to that of *Persistent Bisimulation-based Non Deducibility on Compositions* (PBNDC) on CCS. However, since CCS was not equipped with a type system, the question of whether the translation preserves types could not be addressed. We extend Focardi, Rossi and Sabelfeld's result by showing that a slightly different variant of Milner's translation preserves a *time-insensitive* NI property on the full parallel language, by mapping it again to PBNDC. As a by-product, we formalise a folklore result, namely that Milner's translation preserves a behavioural equivalence on programs. We present a simple type system ensuring PBNDC on CCS, inspired from existing type systems for the π-calculus. Unfortunately, this type system as it stands is too restrictive to grant the expected type preservation result. We suggest a solution to overcome this problem.

Keywords: Noninterference, type systems, parallel imperative languages, process calculi, bisimulation.

1 Introduction

The issue of *secure information flow* has attracted a great deal of interest in recent years, spurred by the spreading of mobile devices and nomadic computation. The question has been studied in some depth both for programming languages (see [26] for a review) and for process calculi [24,8,13,21,11,14,12,5,17,10]. We shall speak of "language-based security" when referring to programming languages, and of "process-based security" when referring to process calculi.

The language-based approach is concerned with secret *data* not being leaked by programs, that is, with the security property of confidentiality. This property is usually formalized via the notion of *noninterference* (NI), stating that secret inputs of programs should not influence their public outputs, since this could allow - at least in principle - a public user to reconstruct secret information.

The process-based approach, on the other hand, is concerned with secret *actions* of processes not being publicly observable. Although bearing a clear analogy with

[1] Work partially supported by the ANR SETI-06-010 grant.

[2] Email: Ilaria.Castellani@sophia.inria.fr

the language-based approach - security levels are assigned in both cases to information carriers, respectively variables and channels - the process-based approach does not rely on quite the same simple intuition. Indeed, there are several choices as to what an observer can gather by communicating with a process. This is reflected in the variety of NI properties proposed for process calculi, mostly based on trace equivalence, testing or bisimulation (cf [8] for a review). In general, these properties do not clearly distinguish between the flow of data and the flow of control, which are closely intertwined in process calculi. Let us consider some examples.

In the calculus CCS with value passing [18], an input process $a(x).P$ receives a value v on channel a and then becomes $P\{v/x\}$. Symmetrically, an output process $\overline{a}\langle e\rangle.P$ emits the value of expression e on channel a and then becomes P. Then a typical insecure data flow is the following, where subscripts indicate the security level of channels (h meaning "high" or "secret", and ℓ meaning "low" or "public"):

$$get_h(x).\overline{put}_\ell\langle x\rangle$$

Here a value received on a high channel is retransmitted on a low channel. Since the value for x may be obtained from some high external source, this process is considered insecure. However, there are other cases where low output actions carry no data, or carry data that do not originate from a high source, as in:

$$get_h(x).\overline{a}_\ell \qquad c_h.\overline{put}_\ell\langle v\rangle \qquad get_h(x).\overline{put}_\ell\langle v\rangle$$

where a_ℓ, c_h are channels without parameters and v is a constant value. Although these processes do not directly transfer data from high to low level, they can be used to implement indirect insecure flows, as in the following process (where x is a boolean and actions on channels c_h and d_h are restricted and thus not observable):

$$P = ((get_h(x).\,\texttt{if}\ x\ \texttt{then}\ \overline{c}_h\ \texttt{else}\ \overline{d}_h) \mid (c_h.\overline{put}_\ell\langle 0\rangle\ +\ d_h.\overline{put}_\ell\langle 1\rangle)) \setminus \{c_h, d_h\}$$

The above examples suggest a simple criterion for enforcing noninterference on CCS, namely that high actions should not be followed by low actions. Admittedly, this requirement is very strong. However, it may serve, and indeed has been used, as a basis for defining *security type systems* for process calculi.

In the language-based approach, theoretical results have often lead to the design of tools for verifying security properties and to the development of secure implementations. Most of the languages examined so far have been equipped with a type system or some other tool to enforce the desired security property [19,20,23,22].

By contrast, the process-based approach has remained at a more theoretical level. Type systems for variants of the π-calculus, which combine the control of security with other correctness concerns, have been proposed by Hennessy et al. in [11,12] and by Honda et al. in [13,14]. A purely security type system for the π-calculus was presented by Pottier in [21]. More recently, different security type systems for the π-calculus were studied by Crafa and Rossi [10] and by Kobayashi [17] (this last work also provides a type inference algorithm). Other static verification methods have been proposed for a variant of CCS in [5].

We address the question of unifying the language-based and process-based approaches, by relating both their security notions and the associated type systems. A first step in this direction was taken by Honda, Vasconcelos and Yoshida in [13], where a parallel imperative language was embedded into a typed π-calculus. This work was pursued by Honda and Yoshida in [14], where more powerful languages,

both imperative and functional, were considered. In [9], Focardi, Rossi and Sabelfeld showed that a variant of Milner's translation of a sequential imperative language into CCS preserves a *time-sensitive* NI property, by mapping it to the property of *Persistent Bisimulation-based Non Deducibility on Compositions* (PBNDC), introduced by Focardi and Rossi in [7]. However, since CCS was not equipped with a security type system, the question of type preservation could not be addressed.

Taking [9] as our starting point, we extend its result by showing that a new variant of Milner's translation preserves a *time-insensitive* NI property on a parallel imperative language, mapping it again to PBNDC. As a by-product, we show that the translation preserves a behavioural equivalence on programs. We also propose a type system for ensuring PBNDC, inspired by the type systems of [21,11,12] for the π-calculus. Unfortunately, this type system is too restrictive as it stands to reflect any of the known type systems for the source language. However, it can be used as a basis to derive a suitable type system, which is briefly sketched here.

The rest of the paper is organised as follows. In Section 2 we recall the definitions of BNDC and PBNDC for CCS and we present a type system ensuring the latter. In Section 3 we introduce the parallel imperative language and the time-insensitive NI property for it; we then propose an adaptation of Milner's translation of this language into CCS, and show that it preserves our NI property. We conclude with a discussion about type preservation. Proofs are omitted and may be found in [6].

2 A simple security type system for CCS

In this section we present a security type system for CCS, inspired by those proposed for the π-calculus by Pottier [21] and by Hennessy and Riely [11,12]. We prove that this type system ensures the property of *Persistent Bisimulation-based Non Deducibility on Compositions* (PBNDC), introduced by Focardi and Rossi in [7].

2.1 The process calculus CCS

Our chosen process calculus is CCS with value passing and guarded sums. We start by recalling the main definitions. We assume a countable set of channels or names $\mathcal{N}$, ranged over by a, b, c, with the usual notational conventions for input and output. Similarly, let Var be a countable set of variables, disjoint from $\mathcal{N}$ and ranged over by x, y, z, and Val be the set of data values, ranged over by v, v'. We define Exp, ranged over by e, e', to be the set of boolean and arithmetic expressions built from values and variables using the standard total operations. Finally, we let $val : Exp \to Val$ be the evaluation function for expressions, satisfying $val(v) = v$ for any value v. We will use the notation $\vec{x}$ (resp. $\vec{v}$ or $\vec{e}$) to denote a sequence $\langle x_1, \ldots, x_n \rangle$ (resp. a sequence $\langle v_1, \ldots, v_n \rangle$ or $\langle e_1, \ldots, e_n \rangle$).

The syntax of process *prefixes*, ranged over by π, π', is given by:

$$\pi ::= a(x) \mid \overline{a}\langle e \rangle \mid a \mid \overline{a}$$

Simple prefixes of the form a and $\overline{a}$ will be used in examples but omitted from our technical treatment, since they are a simpler case of $a(x)$ and $\overline{a}\langle e \rangle$.

To define recursive processes, we assume a countable set $\mathcal{I} = \{A, B, \ldots\}$ of parametric process identifiers, each of which is supposed to have a fixed arity. We

then define the set of *parametric terms*, ranged over by T, T', as follows:

$$T ::= A \;\mid\; (\mathtt{rec}\, A(\vec{x})\,.\,P)$$

where P is a CCS process, as defined next.

A term $(\mathtt{rec}\, A(\vec{x})\,.\,P)$ is supposed to satisfy some standard requirements: (1) all variables in $\vec{x}$ are distinct; (2) the length of $\vec{x}$ is equal to the arity of A; (3) all free variables of P belong to $\vec{x}$; (4) no free process identifier other than A occurs in P; (5) recursion is guarded: all occurrences of A in P appear under a prefix.

The set $\mathcal{P}r$ of *processes*, ranged over by P, Q, R, is given by the syntax:

$$P, Q ::= \textstyle\sum_{i\in I} \pi_i.P_i \;\mid\; (P \mid Q) \;\mid\; (\nu a)\, P \;\mid\; T(\vec{e})$$

where I is an indexing set. We use $\mathbf{0}$ as an abbreviation for the empty sum $\sum_{i\in\emptyset} \pi_i.P_i$. Also, we abbreviate a unary sum $\sum_{i\in\{1\}} \pi_i.P_i$ to $\pi_1.P_1$ and a binary sum $\sum_{i\in\{1,2\}} \pi_i.P_i$ to $(\pi_1.P_1 + \pi_2.P_2)$. In a process $A(\vec{e})$ or $(\mathtt{rec}\, A(\vec{x})\,.\,P)(\vec{e})$, the length of $\vec{e}$ is assumed to be equal to the arity of A. Finally, if $\vec{a} = \langle a_1, \ldots, a_n\rangle$, with $a_i \neq a_j$ for $i \neq j$, the term $(\nu a_1)\cdots(\nu a_n)\, P$ is abbreviated to $(\nu\vec{a})\, P$. If $K = \{a_1, \ldots, a_n\}$, we sometimes render $(\nu\vec{a})\, P$ simply as $(\nu K)\, P$, or use the original CCS notation $P\setminus K$, especially in examples.

The set of free variables (resp. free process identifiers) of process P will be denoted by $\mathit{fv}(P)$ (resp. $\mathit{fid}(P)$). We use $P\{v/x\}$ for the substitution of the variable x by the value v in P. Also, if $\vec{x} = \langle x_1, \ldots, x_n\rangle$ and $\vec{v} = \langle v_1, \ldots, v_n\rangle$, we denote by $P\{\vec{v}/\vec{x}\}$ the substitution of each variable x_i by the value v_i in P. Finally, $P\{T/A\}$ stands for the substitution of the parametric term T for the identifier A in P.

The semantics of processes is given by labelled transitions of the form $P \xrightarrow{\alpha} P'$. Transitions are labelled by *actions* α, β, γ, which are elements of the set:

$$\mathit{Act} \stackrel{\text{def}}{=} \{av : a \in \mathcal{N},\, v \in \mathit{Val}\} \cup \{\bar{a}v : a \in \mathcal{N},\, v \in \mathit{Val}\} \cup \{\tau\}$$

The subject of a prefix is defined by $\mathit{subj}(a(x)) = \mathit{subj}(\overline{a}\langle e\rangle) = a$, and the subject of an action by $\mathit{subj}(av) = \mathit{subj}(\bar{a}v) = a$ and $\mathit{subj}(\tau) = \tau$. The complementation operation is extended to input and output actions by letting $\overline{av} = \bar{a}v$ and $\overline{\bar{a}v} = av$.

The operational rules for CCS processes are recalled in Figure 1. A nondeterministic sum $\sum_{i\in I} \pi_i.P_i$ executes one summand $\pi_i.P_i$, simultaneously discarding the others. A summand $a(x).\, P_i$ receives a value v on channel a and then replaces it for x in P_i. A summand $\overline{a}\langle e\rangle.\, P_i$ emits the value of expression e on channel a and then becomes P_i. The parallel composition $P \mid Q$ interleaves the executions of P and Q, possibly synchronising them on complementary actions to yield a τ-action. The restriction $(\nu b)P$ behaves like P where actions on channel b are forbidden.

2.2 Security properties for CCS

We review two security properties for CCS: *Bisimulation-based Non Deducibility on Compositions* (BNDC), introduced by Focardi and Gorrieri [8] and reformulated by Focardi and Rossi in [7], and *Persistent Bisimulation-based Non Deducibility on Compositions* (PBNDC), proposed in [7] as a strenghtening of BNDC, better suited to deal with dynamic contexts.

$$\text{(SUM-OP}_1\text{)} \quad \textstyle\sum_{i\in I} \pi_i.P_i \xrightarrow{av} P_i\{v/x\}, \ \text{ if } \ \pi_i = a(x) \ \text{ and } \ v \in Val$$

$$\text{(SUM-OP}_2\text{)} \quad \textstyle\sum_{i\in I} \pi_i.P_i \xrightarrow{\overline{a}v} P_i\,, \ \text{ if } \ \pi_i = \overline{a}\langle e\rangle \ \text{ and } \ val(e) = v$$

$$\text{(PAR-OP}_1\text{)} \quad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \text{(PAR-OP}_2\text{)} \quad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\text{(PAR-OP}_3\text{)} \quad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\overline{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \text{(RES-OP)} \quad \frac{P \xrightarrow{\alpha} P' \quad b \neq subj(\alpha)}{(\nu b)P \xrightarrow{\alpha} (\nu b)P'}$$

$$\text{(REC-OP)} \quad \frac{P\{\vec{v}/\vec{x}\}\{\,(\texttt{rec}\ A(\vec{x})\,.\,P)\,/\,A\,\} \xrightarrow{\alpha} P' \qquad \vec{v} = val(\vec{e})}{(\texttt{rec}\ A(\vec{x})\,.\,P)(\vec{e}) \xrightarrow{\alpha} P'}$$

Fig. 1. Operational Semantics of CCS Processes

We start by recalling the definition of weak bisimulation. We adopt the usual notational conventions:

- For any $\alpha \in Act$, let $P \stackrel{\alpha}{\Longrightarrow} P' \stackrel{\text{def}}{=} P \xrightarrow{\tau}{}^{*} \xrightarrow{\alpha} \xrightarrow{\tau}{}^{*} P'$
- For any $\alpha \in Act$, let $P \stackrel{\hat{\alpha}}{\Longrightarrow} P' \stackrel{\text{def}}{=} \begin{cases} P \stackrel{\alpha}{\Longrightarrow} P' & \text{if } \alpha \neq \tau \\ P \xrightarrow{\tau}{}^{*} P' & \text{if } \alpha = \tau \end{cases}$

Definition 2.1 [Weak Bisimulation] A symmetric relation $\mathcal{S} \subseteq (\mathcal{P}r \times \mathcal{P}r)$ is a *weak bisimulation* if $P\ \mathcal{S}\ Q$ implies, for any $\alpha \in Act$:

$$\text{If } P \xrightarrow{\alpha} P' \text{ then there exists } Q' \text{ such that } Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q' \text{ and } P'\ \mathcal{S}\ Q'.$$

P and Q are *weakly bisimilar*, $P \approx Q$, if $P\ \mathcal{S}\ Q$ for some weak bisimulation $\mathcal{S}$.

To set up the scenario for BNDC, we need a few more definitions.

Definition 2.2 [High and low channels] The set $\mathcal{N}$ of channels is partitioned into a subset of high (secret) channels $\mathcal{H}$ and a subset of low (public) channels $\mathcal{L}$.

Input and output actions are then defined to be high or low according to the level of their supporting channel. No security level is given to τ-actions.

Definition 2.3 [Syntactically high processes w.r.t. $\mathcal{H}$]
The set of syntactically high processes with respect to $\mathcal{H}$, denoted $\mathcal{P}r^{\mathcal{H}}_{\text{syn}}$, is the set of processes that contain only channels in $\mathcal{H}$.

The property of *Bisimulation-based Non Deducibility on Compositions* (BNDC) of [8], in its reformulation given by Focardi and Rossi [7], is now defined as follows:

Definition 2.4 [$\mathsf{BNDC}_{\mathcal{H}}$] Let $P \in \mathcal{P}r$ and $\mathcal{H} \subseteq \mathcal{N}$ be the set of high channels. Then P is *secure* with respect to $\mathcal{H}$, $P \in \mathsf{BNDC}_{\mathcal{H}}$, if for every process $\Pi \in \mathcal{P}r^{\mathcal{H}}_{\mathtt{syn}}$, $(\nu\mathcal{H})(P \mid \Pi) \approx (\nu\mathcal{H})P$.

When there is no ambiguity, we write simply BNDC instead of $\mathsf{BNDC}_{\mathcal{H}}$. Let us point out two typical sources of insecurity:

(i) Insecurity may appear when a high name is followed by a low name in P, because in this case the execution of $(\nu\mathcal{H})P$ may block on the high name, making the low name unreachable, while it is always possible to find a high process Π that makes the low name reachable in $(\nu\mathcal{H})(P \mid \Pi)$. Let for instance $P = a_h.\overline{b_\ell}$. Choosing $\Pi = \overline{a_h}$, one obtains $(\nu\mathcal{H})(P \mid \Pi) \not\approx (\nu\mathcal{H})P$.

(ii) Insecurity may also appear when a high name is in conflict with a low name, as in $P = a_h + \overline{b_\ell}$. Here again, taking $\Pi = \overline{a_h}$ one gets $(\nu\mathcal{H})(P \mid \Pi) \not\approx (\nu\mathcal{H})P$, since the first process can do a silent move $\xrightarrow{\tau}$ leading to a state equivalent to $\mathbf{0}$, which the second process cannot match. Note on the other hand that $Q = a_h.\overline{b_\ell} + \overline{b_\ell}$ is secure, because in this case, the synchronisation on channel a_h in $(\nu\mathcal{H})(Q \mid \Pi)$ may be simulated by inaction in $(\nu\mathcal{H})Q$.

In [7], Focardi and Rossi proposed a more robust property than BNDC, which they called *Persistent Bisimulation-based Non Deducibility on Compositions* (PBNDC).

To define PBNDC, a more permissive notion of bisimulation is required, based on a new transition relation $\stackrel{\widetilde{\alpha}}{\Longrightarrow}_{\mathcal{H}}$, defined for any $\alpha \in Act$ by:

$$P \stackrel{\widetilde{\alpha}}{\Longrightarrow}_{\mathcal{H}} P' \quad \stackrel{\text{def}}{=} \quad \begin{cases} P \stackrel{\hat{\alpha}}{\Longrightarrow} P' \text{ or } P \xrightarrow{\tau}{}^{*} P' & \text{if } \ subj(\alpha) \in \mathcal{H} \\ P \stackrel{\hat{\alpha}}{\Longrightarrow} P' & \text{otherwise} \end{cases}$$

Definition 2.5 [Weak bisimulation up-to-high]
A symmetric relation $\mathcal{S} \subseteq (\mathcal{P}r \times \mathcal{P}r)$ is a *weak bisimulation up to high* if $P \ \mathcal{S} \ Q$ implies, for any $\alpha \in Act$:

$$\text{If } P \xrightarrow{\alpha} P' \text{ then there exists } Q' \text{ such that } Q \stackrel{\widetilde{\alpha}}{\Longrightarrow}_{\mathcal{H}} Q' \text{ and } P' \ \mathcal{S} \ Q'.$$

Two processes P, Q are *weakly bisimilar up to high*, written $P \approx_{\mathcal{H}} Q$, if $P \ \mathcal{S} \ Q$ for some weak bisimulation up to high $\mathcal{S}$.

Definition 2.6 [$\mathsf{PBNDC}_{\mathcal{H}}$] Let $P \in \mathcal{P}r$. Then P is said to be *persistently secure* with respect to $\mathcal{H}$, $P \in \mathsf{PBNDC}_{\mathcal{H}}$, if $P \approx_{\mathcal{H}} (\nu\mathcal{H})P$.

The transition relation $\stackrel{\widetilde{\alpha}}{\Longrightarrow}_{\mathcal{H}}$ is used in the definition of PBNDC to allow high moves of P to be matched by (possibly empty) sequences of τ-moves of $(\nu\mathcal{H})P$.

It was shown in [7] that PBNDC is stronger than BNDC, and that requiring PBNDC for P amounts to requiring BNDC for all reachable states of P. All the examples considered above are treated in the same way by BNDC and PBNDC. Examples of secure but not persistently secure processes may be found in [7] or [6].

2.3 A security type system for PBNDC

In this section we present our security type system for CCS and we show that it ensures the PBNDC property. This type system can be viewed as the reduction to CCS of the security type systems proposed for the π-calculus by Pottier [21] and by Hennessy et al. [11,12].

Security levels, ranged over by δ, θ, σ, are defined as usual to form a lattice $(\mathcal{T}, \leq)$, where the order relation $\leq$ stands for "less secret than". Here we assume the lattice to be simply $\{\ell, h\}$, with $\ell \leq h$, to match the partition of the set of channels into $\mathcal{L}$ and $\mathcal{H}$.

A *type environment* Γ is a mapping from channels to security levels, together with a partial mapping from process identifiers to security levels. This mapping is extended to prefixes and visible actions by letting $\Gamma(\pi) = \Gamma(subj(\pi))$ and for any $\alpha \neq \tau$, $\Gamma(\alpha) = \Gamma(subj(\alpha))$. Type judgements for processes have the form $\Gamma \vdash_\sigma P$. Intuitively, $\Gamma \vdash_\sigma P$ means that in the type environment Γ, σ is a *lower bound* on the security level of channels occurring in P. The typing rules are as follows:

$$\text{(Sum)}\quad \frac{\forall i \in I: \quad \Gamma(\pi_i) = \sigma \qquad \Gamma \vdash_\sigma P_i}{\Gamma \vdash_\sigma \sum_{i\in I} \pi_i.P_i} \qquad \text{(Par)}\quad \frac{\Gamma \vdash_\sigma P \qquad \Gamma \vdash_\sigma Q}{\Gamma \vdash_\sigma P \mid Q}$$

$$\text{(Res)}\quad \frac{\Gamma, b:\theta \vdash_\sigma P}{\Gamma \vdash_\sigma (\nu b)P} \qquad \text{(Sub)}\quad \frac{\Gamma \vdash_\sigma P \qquad \sigma' \leq \sigma}{\Gamma \vdash_{\sigma'} P}$$

$$\text{(Rec}_1\text{)}\quad \frac{\Gamma(A) = \sigma}{\Gamma \vdash_\sigma A(\vec{e})} \qquad \text{(Rec}_2\text{)}\quad \frac{\Gamma, A:\sigma \vdash_\sigma P}{\Gamma \vdash_\sigma (\mathtt{rec}\, A(\vec{x})\,.\,P)(\vec{e})}$$

Let us briefly discuss rule (Sum), which is the less standard one. This rule imposes a strong constraint on processes $\sum_{i\in I} \pi_i.P_i$, namely that all prefixes π_i have the same security level σ and that the P_i have themselves type σ. In fact, since each judgement $\Gamma \vdash_\sigma P_i$ may have been derived using subtyping, this means that originally $\Gamma \vdash_{\sigma_i} P_i$ for some σ_i such that $\sigma \leq \sigma_i$. Note that, as expected, $a_h.\overline{b_\ell}$ and $a_h + \overline{b_\ell}$ are not typable. However, the secure process $a_h.\overline{b_\ell} + \overline{b_\ell}$ is not typable either. In [6] we discuss a relaxation of Rule (Sum) which would allow this process to be typed.

We proceed now to establish the soundness of this type system for PBNDC. We state here the most relevant results, referring the reader to [6] for more details.

Theorem 2.7 (Subject reduction)
For any $P \in \mathcal{P}r$, if $\Gamma \vdash_\sigma P$ and $P \xrightarrow{\alpha} P'$ then $\Gamma \vdash_\sigma P'$.

Lemma 2.8 (Confinement)
Let $P \in \mathcal{P}r$ and $\Gamma \vdash_\sigma P$. If $P \xrightarrow{\alpha} P'$ and $\alpha \neq \tau$ then $\Gamma(\alpha) \geq \sigma$.

The key for the soundness proof is the following property of typable programs:

Lemma 2.9 ($\approx_{\mathcal{H}}$– invariance under high actions)

Let $P \in \mathcal{P}r$, $\Gamma \vdash_\sigma P$ and $\mathcal{H} = \{ a \in \mathcal{N} : \Gamma(a) = h \}$. If $P \xrightarrow{\alpha} P'$ and $\Gamma(\alpha) = h$ then $P \approx_{\mathcal{H}} P'$.

Corollary 2.10 (Compositionality of $\approx_{\mathcal{H}}$ for typable programs)
Let $P, Q, R \in \mathcal{P}r$ and Γ be a type environment such that $\Gamma \vdash_\sigma P$, $\Gamma \vdash_\sigma Q$ and $\Gamma \vdash_\sigma R$. Then, if $P \approx_{\mathcal{H}} Q$, also $P \mid R \approx_{\mathcal{H}} Q \mid R$.

Note that $\approx_{\mathcal{H}}$ is not preserved by parallel composition on arbitrary programs, as shown by this example, where $P_i \approx_{\mathcal{H}} Q_i$ for $i = 1, 2$ but $P_1 \mid P_2 \not\approx_{\mathcal{H}} Q_1 \mid Q_2$:

$$P_1 = a_h \qquad Q_1 = \mathbf{0} \qquad P_2 = Q_2 = b_\ell + \overline{a_h}$$

It is easy to see that $P_1 \mid P_2 \not\approx_{\mathcal{H}} Q_1 \mid Q_2$, since $P_1 \mid P_2$ can perform a τ-action which $Q_1 \mid Q_2$ cannot match. Note that P_2 is not typable. In fact P_2 is insecure. Indeed, the property of PBNDC itself is compositional, as shown in [7].

Using the above results, we may show that typability implies PBNDC:

Theorem 2.11 (Soundness)
If $P \in \mathcal{P}r$ and $\Gamma \vdash_\sigma P$ then $P \approx_{\mathcal{H}} (\nu \mathcal{H})P$, where $\mathcal{H} = \{ a \in \mathcal{N} : \Gamma(a) = h \}$.

This concludes, for the time being, our discussion about security and types for CCS.

3 Translating parallel imperative programs into CCS

We focus here on the parallel imperative language studied by Smith and Volpano in [30], which we call PARIMP. Several NI properties and related type systems have been already proposed for this language [27,1,29,4], inspired by the pioneering work of Volpano, Smith et al. [32,30,31]. There exists a well known translation of PARIMP into CCS, presented by Milner in [18]. In [9], Focardi, Rossi and Sabelfeld showed that a variant of this translation preserves – by mapping it to PBNDC – a time-sensitive notion of NI for the sequential fragment of PARIMP. We shall be concerned here with the full language PARIMP, and with a *time-insensitive* NI property for this language. We will prove that this NI property is preserved by a suitable variant of Milner's translation. As a by-product, we will show that our translation also preserves a behavioural equivalence on programs.

3.1 The imperative language PARIMP

In this section we recall the syntax and semantics of the language PARIMP, and we define a time-insensitive NI property for it, inspired by that of [4].

We assume a countable set of variables ranged over by X, Y, Z, a set of values ranged over by V, V', and a set of expressions ranged over by E, E'. Formally, *expressions* are built using total functions $F, G, \ldots$, which we assume to be in a 1 to 1 correspondence with the functions $f, g, \ldots$ used to build CCS expressions:

$$E ::= F(X_1, \ldots, X_n)$$

The set $\mathcal{C}$ of *programs* or *commands*, ranged over by C, D, is defined by:

$$C, D ::= \texttt{nil} \mid X := E \mid C\,;\, D \mid (\texttt{if } E \texttt{ then } C \texttt{ else } D) \mid$$
$$(\texttt{while } E \texttt{ do } C) \mid (C \parallel D)$$

The operational semantics of the language is given in terms of transitions between configurations $\langle C, s\rangle \to \langle C', s'\rangle$ where C, C' are programs and s, s' are *states* or *memories*, that is, mappings from a finite subset of variables to values. These mappings are extended in the obvious way to expressions, whose evaluation is assumed to be terminating and atomic. We use the notation $s[V/X]$ for memory update, $\mapsto$ for the reflexive closure of $\to$, and $\to^*$ for its reflexive and transitive closure. The operational rules for configurations are given in Figure 2. The rules (ParL-Op2) and (ParR-Op2) are introduced, as in [3], to allow every terminated configuration to take the form $\langle \texttt{nil}, s\rangle$.

A configuration $\langle C, s\rangle$ is *well-formed* if $fv(C) \subseteq \texttt{dom}(s)$. It is easy to see, by inspection of the rules, that $\langle C, s\rangle \to \langle C', s'\rangle$ implies $fv(C') \subseteq fv(C)$ and $\texttt{dom}(s') = \texttt{dom}(s)$. Hence well-formedness is preserved by execution.

As for CCS, we assume variables to be partitioned into a set of *low variables* L and a set of *high variables* H. In examples, we will use the subscripts L and H for variables belonging to the sets L and H, respectively. We may now introduce the notions of low-equality and low-bisimulation.

Definition 3.1 [L-Equality] Two memories s and t are *L-equal*, written $s =_L t$, if $\texttt{dom}(s) = \texttt{dom}(t)$ and $(X \in \texttt{dom}(s) \cap L \;\Rightarrow\; s(X) = t(X))$.

Definition 3.2 [L-Bisimulation]
A symmetric relation $\mathcal{S} \subseteq (\mathcal{C} \times \mathcal{C})$ is a *L-bisimulation* if $C\,\mathcal{S}\,D$ implies, for any pair of states s and t such that $s =_L t$ and $\langle C, s\rangle$ and $\langle D, t\rangle$ are well-formed:

$$\text{If } \langle C, s\rangle \to \langle C', s'\rangle, \text{ then there exist } D', t' \text{ such that}$$
$$\langle D, t\rangle \mapsto \langle D', t'\rangle \text{ and } s' =_L t' \text{ and } C'\,\mathcal{S}\,D'.$$

Two programs C, D are *L-bisimilar*, $C \simeq_L D$, if $C\,\mathcal{S}\,D$ for some L-bisimulation $\mathcal{S}$.

Note that the simulating program is required to mimic each move of the first program by either one or zero moves. This notion of low-bisimulation is inspired from [4]. We could have chosen a weaker notion, where $\mapsto$ is replaced by $\to^*$, as proposed in [27]. However our choice allows for a more precise notion of security, which respects "state-traces", as illustrated by Example 3.4.

Definition 3.3 [L-Security] A program C is *L-secure* if $C \simeq_L C$.

When L is clear, we shall speak simply of *low-equality*, *low-bisimulation* and *security*.

Example 3.4 The following program, where $\texttt{loop } D \stackrel{\text{def}}{=} (\texttt{while } tt \texttt{ do } D)$:

$$C = (\texttt{if } X_H = 0 \texttt{ then loop } (Y_L := 0\,;\, Y_L := 1) \texttt{ else loop } (Y_L := 1\,;\, Y_L := 0))$$

is not L-secure since the branches of the conditional cannot simulate each other's moves in one or zero steps. However it would be secure according to the weaker notion of L-bisimulation obtained by replacing $\mapsto$ with $\to^*$ in Definition 3.2.

$$\text{(Assign-Op)}\quad \frac{}{\langle X := E, s\rangle \to \langle \mathtt{nil}, s[s(E)/X]\rangle}$$

$$\text{(Seq-Op1)}\quad \frac{\langle C, s\rangle \to \langle C', s'\rangle}{\langle C; D, s\rangle \to \langle C'; D, s'\rangle} \qquad \text{(Seq-Op2)}\quad \frac{}{\langle \mathtt{nil}; D, s\rangle \to \langle D, s\rangle}$$

$$\text{(Cond-Op1)}\quad \frac{s(E) = tt}{\langle \mathtt{if}\ E\ \mathtt{then}\ C\ \mathtt{else}\ D, s\rangle \to \langle C, s\rangle}$$

$$\text{(Cond-Op2)}\quad \frac{s(E) \neq tt}{\langle \mathtt{if}\ E\ \mathtt{then}\ C\ \mathtt{else}\ D, s\rangle \to \langle D, s\rangle}$$

$$\text{(While-Op1)}\quad \frac{s(E) = tt}{\langle \mathtt{while}\ E\ \mathtt{do}\ C, s\rangle \to \langle C; \mathtt{while}\ E\ \mathtt{do}\ C, s\rangle}$$

$$\text{(While-Op2)}\quad \frac{s(E) \neq tt}{\langle \mathtt{while}\ E\ \mathtt{do}\ C, s\rangle \to \langle \mathtt{nil}, s\rangle}$$

$$\text{(ParL-Op1)}\quad \frac{\langle C, s\rangle \to \langle C', s'\rangle}{\langle C \parallel D, s\rangle \to \langle C' \parallel D, s'\rangle} \qquad \text{(ParL-Op2)}\quad \frac{}{\langle \mathtt{nil} \parallel D, s\rangle \to \langle D, s\rangle}$$

$$\text{(ParR-Op1)}\quad \frac{\langle D, s\rangle \to \langle D', s'\rangle}{\langle C \parallel D, s\rangle \to \langle C \parallel D', s'\rangle} \qquad \text{(ParR-Op2)}\quad \frac{}{\langle C \parallel \mathtt{nil}, s\rangle \to \langle C, s\rangle}$$

Fig. 2. Operational Semantics of PARIMP

3.2 Milner's translation of PARIMP into CCS

We now review Milner's translation of the language PARIMP into CCS [18]. This translation makes use of two new constructs of CCS, renaming and conditional, whose semantics we assume to be known (see [18], or the full paper [6]).

First, *registers* are introduced to model the store. For each program variable X, the associated register Reg_X, parameterised by the value it contains, is defined by:

$$Reg_X(v) \stackrel{\text{def}}{=} put_X(x).Reg_X(x) + \overline{get_X}\langle v\rangle.Reg_X(v)$$

The translation $[\![s]\!]$ of a state s is then a *pool of registers*, given by :

$$[\![s]\!] = Reg_{X_1}(s(X_1)) \mid \cdots \mid Reg_{X_n}(s(X_n)) \qquad \text{if } \mathtt{dom}(s) = \{X_1, \ldots, X_n\}$$

The translation $[\![E]\!]$ of an expression $E = F(X_1, \ldots, X_n)$ is a process which fetches the values of registers $Reg_{X_1} \ldots, Reg_{X_n}$ into the variables $x_1, \ldots, x_n$ and then transmits over a special channel $\mathtt{res}$ the result of evaluating $f(x_1, \ldots, x_n)$, where f is the CCS function corresponding to the PARIMP function F:

$$[\![F(X_1, \ldots, X_n)]\!] = get_{X_1}(x_1).\cdots.get_{X_n}(x_n).\overline{\mathtt{res}}\langle f(x_1, \ldots, x_n)\rangle.\mathbf{0}$$

The channel `res` is used by the auxiliary operator *Into*, defined by:

$$P\ \mathit{Into}\,(x)\ Q \stackrel{\text{def}}{=} (P \mid \texttt{res(x)}.\,Q)\backslash\texttt{res}$$

To model sequential composition, a special channel `done` is introduced, on which processes signal their termination. Channel `done` is used by the auxiliary operators *Done*, *Before* and *Par*, defined as follows, assuming d, d_1, d_2 to be new names:

$$\mathit{Done} \stackrel{\text{def}}{=} \overline{\texttt{done}}.\,\mathbf{0}$$

$$C\ \mathit{Before}\ D \stackrel{\text{def}}{=} (C[d/\texttt{done}] \mid d.\,D)\backslash d$$

$$C_1\ \mathit{Par}\ C_2 \stackrel{\text{def}}{=} (C_1[d_1/\texttt{done}] \mid C_2[d_2/\texttt{done}] \mid (d_1.\,d_2.\mathit{Done} + d_2.\,d_1.\mathit{Done}))\backslash\{d_1, d_2\}$$

The translation of commands is then given by:

$$[\![\texttt{nil}]\!] = \mathit{Done} \qquad [\![X := E]\!] = [\![E]\!]\ \mathit{Into}\,(x)\ (\overline{\mathit{put}_X}\langle x\rangle.\,\mathit{Done})$$

$$[\![C\,;\,D]\!] = [\![C]\!]\ \mathit{Before}\ [\![D]\!] \qquad [\![(C_1 \parallel C_2)]\!] = [\![C_1]\!]\ \mathit{Par}\ [\![C_2]\!]$$

$$[\![(\texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2)]\!] = [\![E]\!]\ \mathit{Into}\,(x)\ \ (\texttt{if}\ x\ \texttt{then}\ [\![C_1]\!]\ \texttt{else}\ [\![C_2]\!])$$

$$[\![(\texttt{while}\ E\ \texttt{do}\ C)]\!] = W \stackrel{\text{def}}{=} [\![E]\!]\ \mathit{Into}\,(x)\ (\texttt{if}\ x\ \texttt{then}\ [\![C]\!]\ \mathit{Before}\ W\ \texttt{else}\ \mathit{Done})$$

Finally, the translation of a well-formed configuration $\langle C, s\rangle$ is defined by:

$$[\![\langle C, s\rangle]\!] = (\ [\![C]\!] \mid [\![s]\!]\ )\setminus \mathit{Acc}_s \cup \{\texttt{done}\}$$

where $\mathit{Acc}_s \stackrel{\text{def}}{=} \{\ \mathit{get}_X, \mathit{put}_X \ \mid\ X \in \texttt{dom}(s)\ \}$ is the *access sort* of state s.

As noted by Milner in [18], the above translation does not preserve the atomicity of assignment statements. Consider the program $C = (X := X + 1 \parallel X := X + 1)$. The translation of C is:

$$\begin{aligned}[\![C]\!] = {} & (\ (\mathit{get}_X(x).\,\overline{\texttt{res}}\langle x+1\rangle \mid \texttt{res}(y).\,\overline{\mathit{put}_X}\langle y\rangle.\,\overline{d_1})\ \backslash\texttt{res} \\ & \mid (\mathit{get}_X(x).\,\overline{\texttt{res}}\langle x+1\rangle \mid \texttt{res}(y).\,\overline{\mathit{put}_X}\langle y\rangle.\,\overline{d_2})\ \backslash\texttt{res} \\ & \mid (d_1.\,d_2.\mathit{Done} + d_2.\,d_1.\mathit{Done})\ )\setminus\{d_1, d_2\}\end{aligned}$$

Here the second get_X action may be executed before the first $\overline{\mathit{put}_X}$ action. This means that the same value v_0 may be read for X in both assignments, and thus the same value $v_1 = v_0 + 1$ may be assigned twice to X. Hence, while C only produces the final value $v_2 = v_0 + 2$ for X, $[\![C]\!]$ may also produce the final value $v_1 = v_0 + 1$.

It is then easy to see that the translation does not preserve security. Let $C_L = (X_L := X_L + 1 \parallel X_L := X_L + 1)$ and $D_L = (X_L := X_L + 1\,;\, X_L := X_L + 1)$. Let now $\widehat{C} = (\texttt{if}\ z_H = 0\ \texttt{then}\ C_L\ \texttt{else}\ D_L)$. Then $\widehat{C}$ is secure, but $[\![\widehat{C}]\!]$ is not.

It may be shown with similar examples [6] that, in order for the translation to preserve security, it should also forbid the overlapping of assignments to different variables, as well as the overlapping of assignments with expression evaluation. To prevent such overlappings, we introduce a global semaphore for the whole store:

$$\mathit{Sem} \stackrel{\text{def}}{=} \texttt{lock}.\,\texttt{unlock}.\,\mathit{Sem}$$

The translation of the assignment command then becomes:

$$[\![X := E]\!] = \overline{\texttt{lock}}.\,[\![E]\!]\ \mathit{Into}\,(x)\ (\overline{\mathit{put}_X}\langle x\rangle.\,\overline{\texttt{unlock}}.\,\mathit{Done})$$

The translation of conditionals and loops is adapted in a similar way, by replacing $[\![E]\!]$ with $[\![E]\!]_{at}$, the *atomic translation* of expression E, defined as follows:

$$[\![F(X_1,\ldots,X_n)]\!]_{at} = \overline{\texttt{lock}}.\, getseq_{\vec{X}}(\vec{x}).\, \overline{\texttt{res}}\langle f(\vec{x})\rangle.\, \overline{\texttt{unlock}}.\, \mathbf{0}$$

where $getseq_{\vec{X}}(\vec{x})$ is an abbreviation for the sequence $get_{X_1}(x_1).\cdots.get_{X_n}(x_n)$.

The translation of configurations $\langle C, s\rangle$ is then modified accordingly:

$$[\![\langle C, s\rangle]\!] = (\, [\![C]\!] \mid [\![s]\!] \mid Sem\,) \setminus Acc_s \cup \{\texttt{done}, \texttt{lock}, \texttt{unlock}\}$$

3.3 The translation preserves security

In this section we show that the translation just described preserves security. This result will be based, as usual, on an operational correspondence between programs (or more exactly, configurations) in the source language and their images in the target language. In order to relate the behaviour of a configuration $\langle C, s\rangle$ with that of its image $[\![\langle C, s\rangle]\!] = (\, [\![C]\!] \mid [\![s]\!] \mid Sem\,) \setminus Acc_s \cup \{\texttt{done}, \texttt{lock}, \texttt{unlock}\}$, we must provide a means to observe the changes performed by $[\![C]\!]$ on $[\![s]\!]$ in CCS [3]. To this end we introduce, as in [25] and [9], special channels dedicated to the exchange of data between processes and the environment, which we call *in* and *out*: the environment uses channel in_X to feed a new value into register Reg_X, and channel out_X to retrieve the current value of Reg_X.

The definition of registers is then adapted to account for the new actions. Each Reg_X in our translation is replaced by the *observable register* $OReg_X$ defined by:

$$\begin{aligned} OReg_X(v) \stackrel{\text{def}}{=}\ & put_X(x).\, OReg_X(x) \ +\ \overline{get_X}\langle v\rangle.\, OReg_X(v) \ + \\ & \overline{\texttt{lock}}.\,(\, in_X(x).\, \overline{\texttt{unlock}}.\, OReg_X(x) \ +\ \overline{\texttt{unlock}}.\, OReg_X(v)\,) \ + \\ & \overline{\texttt{lock}}.\,(\, \overline{out_X}\langle v\rangle.\, \overline{\texttt{unlock}}.\, OReg_X(v) \ +\ \overline{\texttt{unlock}}.\, OReg_X(v)\,) \end{aligned}$$

Here the locks around the $in_X(x)$ and $\overline{out_X}\langle v\rangle$ prefixes are used to prevent the environment from accessing the register while this is being used by some process. Note that after committing to communicate with the environment by means of a $\overline{\texttt{lock}}$ action, an observable register can always withdraw its commitment by doing an $\overline{\texttt{unlock}}$ action, and get back to its initial state.

Notation: Let Env be the set $\{in_X, out_X \mid X \in Var\}$. We define then the set of *environmental actions* to be $Act_{Env} \stackrel{\text{def}}{=} \{\alpha \in Act \mid subj(\alpha) \in Env\}$.

As in [9], we define now labelled transitions $\xrightarrow{in_X v}$ and $\xrightarrow{\overline{out}_X v}$ for configurations [4]:

$$\text{(In-Op)}\quad \frac{X \in \texttt{dom}(s)}{\langle C, s\rangle \xrightarrow{in_X v} \langle C, s[v/X]\rangle} \qquad \text{(Out-Op)}\quad \frac{s(X) = v}{\langle C, s\rangle \xrightarrow{\overline{out_X} v} \langle C, s\rangle}$$

We also extend τ-transitions to configurations by letting:

$$\langle C, s\rangle \xrightarrow{\tau} \langle C', s'\rangle \quad \Leftrightarrow_{\text{def}} \quad \langle C, s\rangle \to \langle C', s'\rangle$$

We may now define weak labelled transitions $\langle C, s\rangle \stackrel{\alpha}{\Longrightarrow} \langle C', s'\rangle$ on configurations, where $\alpha \in Act_{Env} \cup \{\tau\}$, exactly in the same way as for CCS processes.

[3] Note that, as it stands, the translation maps any configuration $\langle C, s\rangle$ to an unobservable CCS process.
[4] From now on we use v, v' also for PARIMP values, assuming them to coincide with CCS values.

The operational correspondence between well-formed configurations $\langle C, s\rangle$ and their images in CCS is then given by the following two Lemmas:

Lemma 3.5 (Program transitions are preserved by the translation)

Let $\langle C, s\rangle$ be a well-formed configuration and $\alpha \in Act_{Env}$. Then:

(i) *If $\langle C, s\rangle \xrightarrow{\alpha} \langle C', s'\rangle$, then there exists P such that $[\![\langle C, s\rangle]\!] \stackrel{\alpha}{\Longrightarrow} P = [\![\langle C', s'\rangle]\!]$*

(ii) *If $\langle C, s\rangle \xrightarrow{\tau} \langle C', s'\rangle$, then there exists P such that $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\Longrightarrow} P \approx [\![\langle C', s'\rangle]\!]$*

Lemma 3.6 (Process transitions are reflected by the translation)

Let $\langle C, s\rangle$ be a well-formed configuration and $\alpha \in Act_{Env}$. Then:

(i) *If $[\![\langle C, s\rangle]\!] \stackrel{\alpha}{\Longrightarrow} P$, then there exist C', s' such that $\langle C, s\rangle \stackrel{\alpha}{\Longrightarrow} \langle C', s'\rangle$ and $P \approx [\![\langle C', s'\rangle]\!]$.*

(ii) *If $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\Longrightarrow} P$, then either $[\![\langle C, s\rangle]\!] \approx P$ or there exist C', s' such that $\langle C, s\rangle \stackrel{\tau}{\Longrightarrow} \langle C', s'\rangle$ and $P \approx [\![\langle C', s'\rangle]\!]$.*

Suppose that channels get_X and put_X have the same security level as variable X, that channels $\mathtt{lock}, \mathtt{unlock}, \mathtt{res}$ and $\mathtt{done}$ have level h, and that renaming preserves security levels. We may then show that the translation preserves security.

Theorem 3.7 (Security is preserved by the translation)

If C is a L-secure program, then for any state s such that $\langle C, s\rangle$ is well-formed, $[\![\langle C, s\rangle]\!]$ satisfies $\mathsf{PBNDC}_{\mathcal{H}}$, where $\mathcal{H} \stackrel{\text{def}}{=} \{get_X, put_X, in_X, out_X \mid X \in H\} \cup \{\mathtt{lock}, \mathtt{unlock}, \mathtt{res}, \mathtt{done}\}$.

As a by-product, we show that the translation preserves the behavioural equivalence which is obtained from low bisimilarity by assuming all program variables to be low. If $H = \emptyset$, it is easy to see that L-bisimilarity reduces to the following:

Definition 3.8 [Behavioural equivalence on programs]
A symmetric relation $\mathcal{S} \subseteq (\mathcal{C} \times \mathcal{C})$ is a *program bisimulation* if $C\,\mathcal{S}\,D$ implies, for any state s such that $\langle C, s\rangle$ and $\langle D, s\rangle$ are well-formed:

If $\langle C, s\rangle \to \langle C', s'\rangle$, then there exists D' such that $\langle D, s\rangle \mapsto \langle D', s'\rangle$ and $C'\,\mathcal{S}\,D'$.

Two programs C and D are *behaviourally equivalent*, written $C \simeq D$, if $C\,\mathcal{S}\,D$ for some program bisimulation $\mathcal{S}$.

Theorem 3.9 (Behavioural equivalence is preserved by the translation)

If $C \simeq D$, then for any s such that $\langle C, s\rangle$ is well-formed, $[\![\langle C, s\rangle]\!] \approx [\![\langle D, s\rangle]\!]$.

To sum up, our translation preserves two semantic notions: behavioural equivalence and security.

3.4 The translation does not preserve security types

In this section, we show that the type system presented in Section 2 is not reflected by our translation, and we suggest a solution to overcome this problem.

Consider the program $C = (X_H := X_H + 1\,;\, Y_L := Y_L + 1)$, which is typable in the type systems of [30,27,29,4]. This program is translated to the process $[\![C]\!]$:

$$(\nu d)\,(\,\overline{\texttt{lock}}.\,(\nu\,\texttt{res}_1)\,(get_{X_H}(x).\,\overline{\texttt{res}_1}\langle x+1\rangle \mid \texttt{res}_1(z_1).\,\overline{put_{X_H}}\langle z_1\rangle.\,\overline{\texttt{unlock}}.\,\bar{d}) \mid$$
$$d.\,\overline{\texttt{lock}}.\,(\nu\,\texttt{res}_2)\,(get_{Y_L}(y).\,\overline{\texttt{res}_2}\langle y+1\rangle \mid \texttt{res}_2(z_2).\,\overline{put_{Y_L}}\langle z_2\rangle.\,\overline{\texttt{unlock}}.\,\overline{\texttt{done}})\,)$$

Now, it is easy to see that there is no assignment of security levels for the channels $\texttt{lock}, \texttt{unlock}$ and d which would allow $[\![C]\!]$ to be typed. Note that giving all these channels level h, as we did in the previous section, would not be appropriate here: if d and $\texttt{lock}$ had level h, then the second component of $[\![C]\!]$ would not be typable.

A possible solution to this problem is to relax the type system by treating more liberally actions like $\overline{\texttt{lock}}$, $\overline{\texttt{unlock}}$ and d (and hence $\texttt{done}$), which carry no values and are restricted. The idea, borrowed from [16,14,15,33,17], is that these actions are *data flow irrelevant* insofar as they are guaranteed to occur, since in this case their occurrence does not bring any information. The typing rule (Sum) may then be made less restrictive for these actions, while keeping their security level to h.

Note indeed that, as a consequence of Lemma 3.6, actions $\overline{\texttt{lock}}$ and $\overline{\texttt{unlock}}$ are eventually enabled from any state of a process $[\![\langle C, s\rangle]\!]$. The situation is not as simple as concerns action $\texttt{done}$, as its occurrence may be prevented by divergence or deadlock. Note however that deadlock cannot arise in a process $[\![\langle C, s\rangle]\!]$, because the source configuration $\langle C, s\rangle$ can only contain livelocks, due to busy waiting and thus to while loops. By imposing restrictions on the use of loops in programs (similar to those of [30]), one may then enforce the occurrence of $\overline{\texttt{done}}$ in their images.

Acknowledgments

I would like to thank Maria-Grazia Vigliotti for her contribution at an early stage of this work, Frédéric Boussinot for helpful remarks, and the anonymous referees for useful feedback on the submitted version.

References

[1] Johan Agat. Transforming out timing leaks. *Proceedings of POPL '00*, ACM Press, pages 40–53, 2000.

[2] A. Almeida Matos, G. Boudol and I. Castellani. Typing noninterference for reactive programs. *Journal of Logic and Algebraic Programming* 72: 124-156, 2007.

[3] G. Barthe and L. Prensa Nieto. Formally verifying information flow type systems for concurrent and thread systems. In *Proceedings of FMSE'04*, 2004.

[4] G. Boudol and I. Castellani. Noninterference for Concurrent Programs and Thread Systems. *Theoretical Computer Science* 281(1): 109-130, 2002.

[5] A. Bossi, R. Focardi, C. Piazza and S. Rossi. Verifying persistent security properties. *Computer Languages, Systems and Structures* 30(3-4): 231-258, 2004.

[6] I. Castellani. State-oriented noninterference for CCS (complete version). INRIA RR, to appear.

[7] R. Focardi and S. Rossi. Information flow security in dynamic contexts. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, 2002.

[8] R. Focardi and R. Gorrieri. Classification of Security Properties (Part I: Information Flow). In *Foundations of Security Analysis and Design - Tutorial Lectures* (R. Focardi and R. Gorrieri, Eds.), volume 2171 of *LNCS*, Springer, 2001.

[9] R. Focardi, S. Rossi and A. Sabelfeld. Bridging Language-Based and Process Calculi Security. In *Proceedings of FoSSaCs'05*, volume 3441 of *LNCS*, Springer-Verlag, 2005.

[10] S. Crafa and S. Rossi. A theory of noninterference for the π-calculus. In *Proceedings of Symp. on Trustworthy Global Computing TGC'05*, volume 3705 of *LNCS*, Springer-Verlag, 2005.

[11] M. Hennessy and J. Riely. Information flow vs resource access in the asynchronous π-calculus. *ACM TOPLAS* 24(5): 566-591, 2002.

[12] M. Hennessy. The security π-calculus and noninterference. *Journal of Logic and Algebraic Programming* 63(1): 3-34, 2004.

[13] K. Honda, V. Vasconcelos and N. Yoshida. Secure information flow as typed process behavior. In *Proceedings of ESOP'00*, volume 1782 of *LNCS*, pages 180-199, Springer-Verlag, 2000.

[14] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proceedings of POPL'02*, ACM Press, pages 81-92. January, 2002.

[15] N. Yoshida, K. Honda and M. Berger. Linearity and bisimulation. In *Proceedings of FoSSaCs'02*, volume 2303 of *LNCS*, pages 417-433, Springer-Verlag, 2002.

[16] N. Kobayashi, B. Pierce and D. Turner. Linearity and the π-calculus. In *Proceedings of POPL'96*, ACM Press, pages 358-371, 1996.

[17] N. Kobayashi. Type-based Information Flow Analysis for the π-Calculus. *Acta Informatica* 42(4-5): 291-347, 2005.

[18] R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[19] A. Myers. JFlow: practical mostly-static information flow control. In *Proceedings of POPL'99*, ACM Press, pages 228-241, 1999.

[20] A. Myers, L. Zheng, S. Zdancewic, S. Chong and N. Nystrom. Jif: Java information flow. Software release, http://www.cs.cornell.edu/jif, 2001.

[21] F. Pottier. A Simple View of Type-Secure Information Flow in the π-Calculus. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 320–330, 2002.

[22] F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS* 25(1): 117-158, 2003.

[23] V. Simonet. The FlowCaml system: documentation and user manual. INRIA RR n. 0282, 2003.

[24] P. Ryan and S. Schneider. Process algebra and noninterference. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 214–227, 1999.

[25] H. Mantel and A. Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *Journal of Computer Security* 11(4): 615–676, 2003.

[26] A. Sabelfeld and A. C. Myers, Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 211:5-19, 2003.

[27] A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-threaded Programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 200-214, 2000.

[28] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[29] G. Smith. A new type system for secure information flow. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 115–125, 2001.

[30] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. *Proceedings of POPL '98*, ACM Press, pages 355–364, 1998.

[31] D. Volpano and G. Smith. Probabilistic Noninterference in a Concurrent Language. *Journal of Computer Security* 7(2-3): 231–253, 1999.

[32] D. Volpano, G. Smith and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 4(3):167–187, 1996.

[33] S. Zdancewic and A. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation* 15(2-3):209-234, 2002.

A probabilistic scheduler for the analysis of cryptographic protocols[1]

Srečko Brlek*, Sardaouna Hamadou**, John Mullins*,**

* *Lab. LaCIM, Dép. d'Informatique, Université du Québec à Montréal. CP 8888 Succursale Centre-Ville, Montreal (Quebec), Canada, H3C 3P8.*

** *Lab. CRAC, Dép. de Génie Informatique, École Polytechnique de Montréal P.O. Box 6079, Station Centre-ville, Montreal (Quebec), Canada, H3C 3A7.*

Abstract

When modeling cryto-protocols by mean of process calculi which express both nondeterministic and probabilistic behavior, it is customary to view the scheduler as an intruder. It has been established that in this case the traditional scheduler need to be carefully calibrated in order to reflect more accurately the intruder's capabilities to control communication channels. We propose such a class of schedulers through a semantic variant called $\mathrm{PPC}_{\nu\sigma}$, of the Probabilistic Poly-time Calculus (PPC) of Mitchell *et al.* [11] and we illustrate the pertinence of our approach by an extensive study of the *Dining Cryptographers* (DCP) [8] protocol. Along the lines, we define a new characterization of the Mitchell *et al.*'s observational equivalence [11] more suited to take into account any observable trace instead of a single action as it is required in the analysis of the DCP.

Keywords: Process algebra, observational equivalence, probabilistic scheduling, analysis of cryptographic protocols

1 Introduction

Systems that combine both probabilies and nondeterminism are very convenient for modelling probabilistic security protocols. In order to model such systems, some efforts have been undertaken in extending (possibilistic) models based on process algebras such as the π-calculus or CSP, by including probabilities. One distinguish two classes of such models. On one hand, we have all purpose probabilistic models adding probabilities to nondeterministic models [1,6,3]. On the other hand, we have process algebraic frameworks that define probabilistic models to the purpose of making them more suitable to applications in security protocols [11,4,10].

While it's customary to use schedulers for resolving non-determinism in probabilistic systems, scheduling process must be carefully design in order to reflect as

[1] Research partially supported by NSERC grants (Canada)

accurately as possible the intruder's capabilities to control the communication network without controlling the internal reactions of the system. Indeed, consider the protocol $\overline{c}(a).\mathbf{0}|\overline{c}(b).\mathbf{0}$ transmitting the messages a and b over c and the intruder $c(x).\mathbf{0}$ eavesdropping on this channel. As the protocol is purely non deterministic, the probability that the intercepted message to be either a or b should be the same. A scheduler that could fix an arbitrary probabilistic distribution to these two messages could also force the protocol to transmit either a or b. Such schedulers are too stong and hence should not be admissible. However restricting the power of schedulers must be carefully done as well, otherwise it could result in too weak adversaries. For example, forcing schedulers to give priority to internal actions makes internal actions completely invisible to attackers. Indeed, an intruder is then unable to distinguish a proccess P from another process which can do some internal action and then behaves like P. But now consider the following process:

$$P = \nu c'(c(x).\overline{c'}(x).\mathbf{0}|\overline{c'}(1).\mathbf{0}|c'(y).[y = 0]\overline{c}(\mathit{secret}).\mathbf{0}).$$

In this obviously unsecure protocol, an intruder could send 0 to P over c and so, allow P to publish the secret. Such a flaw will never be detected in a semantics giving priority to internal actions since in that case, P will never broadcast *secret* on the public channel c.

Contribution. Our contribution is threefold. Firstly, we define a semantic variant of the *Probabilistic Polynomial-time Process Calculus PPC* [11] (Section 2), called $\mathrm{PPC}_{\nu\sigma}$ in order to cope with the problem of caracterizing the intruders's power. Contrarily to most of probabilistic models, our operational semantics does not normalize probabilities. The reason is that normalizing has the effect to carry off the control on its own actions to the intruder. Consider the process $P = \overline{c}(m).Q_1|\overline{c}(m).Q_2$. Depending on whether P represents a protocol or an intruder, the scheduling of a component is respectively equiprobable or arbitrarily chosen by the intruder. A solution might be to discriminate semantically between a protocol and an intruder but it rapidly appears quite intricate since synchronization actions could commit both. We propose here a simpler solution to this problem. It consists to equip the intruder with an attack strategy i.e., a selection process called *external scheduler* (Section 2.3) allowing him to choose at each evaluation step the next action to perform. This scheduling is carefully designed to reflect as accurately as possible the intruder's real capabilities,that is to control the communication network without controlling internal reactions of the system under its stimuli.

Secondly, we reformulate (Section 3) the observational equivalence of [11] into a more amenable form to take into account any observable trace instead of a single step.

Finally, to illustrate the pertinence of our approach, we conclude the paper by an extensive case study (Section 4): the analysis of the Chaum's *Dining Cryptographers* protocol [8]. We give a probabilistic version of the possibilistic specification of *anonymity* property due to Schneider and Sidiropoulos [12], and prove that restricting too much the scheduler's power may lead to very weak models that cannot detect a flawed specification of the protocol.

Related work. The technical precursor of our framework is the process calculus of Mitchell *et al.* [11]. Though that any of the models [1,6,3,11,4,10] or any similar

framework could have been an interesting starting point, the framework of Mitchell *et al.* [11] appears appropriate for the following reasons. Although it is a formal model, it is yet closed to computational setting of modern cryptography since it works directly on the cryptographic level. Indeed it defines an extension of the CCS process algebra with finite replication and probabilistic polynomial-time terms denoting cryptographic primitives. It turns that these probabilistic polynomial functions are useful to model the probabilistic behaviour of security systems. Unlike formalisms as [6,4,10], scheduling is probabilistic, reflecting so in a better way the ability of the attacker to control the communication network. Finally it also appears as a natural formal framework to capture and reason about a variety of fundamental cryptographic notions.

The problem of characterizing the schedulers' power has been recently considered in [5,7,9]. In [5] the authors treated the problem of too powerful *adversarial scheduler* in the context of systems modelled in a *Probabilistic I/O Automata* framework. They resticted the scheduler by defining two levels of schedulers. A high-level scheduler called *adversarial scheduler* is a component of the system and controls the communication network, i.e. it schedules public channels. This component has limited knowledge of the behaviour of other components in the system: their internal choices and secret information are hidden. On the other hand, a low-level scheduler called *tasks scheduler* resolves the remaining non-determinism by a *task schedule sequence*. These tasks are equivalence classes of actions that are independent of the high-level scheduler choices. We believe that these tasks may correspond to our "strategically equivalent actions".

In Garcia *et. al.* [9] is addressed a dual problem to the one we have considered here, namely the problem that arises when traditional schedulers are too powerful. In the context of security protocols modelled by probabilistic automata, they defined a probabilistic scheduler that imposes locally a probability distribution on the possible non-deterministic next transitions. Unlike our scheduler, it is history dependent since it defines equiprobable paths and it is not stochastic, and hence may halt the execution at any time. Roughly speaking, admissible schedulers are defined w.r.t bisimulation equivalence: any observably trace equivalent paths are equiprobably scheduled and lead to bisimilar states.

Another recent paper on the scheduling issue is presented in [7]. Unlike our scheduler and the one of [9] which are defined on the semantic level, [7] proposes a framework in which schedulers are defined and controlled on the syntactic level. They make random choices in the protocol invisible to the adversary. Note that we achieve the same goal thanks to the operational semantics `Eval` rule which reduces unblocked processes and our strategically equivalent classes of actions. However these papers ([5,7]) are too recent and a more investigation is needed in order to determine how each approach may benefit from and to another.

Finally an alternate approach is proposed in [4,10]. Instead of scheduling a single action (like ours) or a path (like the one of [9]), a process is scheduled. The problem of discriminating protocol's actions and the intruder's ones, and privileging or not internal actions is meaningless in these models because scheduling is implicitly included in the specification. In other words, the analyzer is the one who determines when the control passes from the protocol to the attacker. Let us explain this last

point. Consider the protocol $P = \nu(c)(\overline{c}(1).|c(x).\overline{c'}(0).)$ which, after an internal communication, outputs 0 on the public channel c'. In the frameworks of [4,10], it may be specified in two different manners

$$P_1 = \nu(c)(start().\overline{c}(1).\mathbf{0}|c(x).\overline{c'}(0).\mathbf{0})$$

and

$$P_2 = \nu(c)(start().\overline{c}(1).\mathbf{0}|c(x).\overline{contr2Intr}().getContr().\overline{c'}(0).\mathbf{0})$$

depending on whether we want to make the internal action completely invisible to the attacker or not. In that way the user has total freedom and can eliminate undesirable schedulers at the specification level. The drawback is that the protocol analyzer who has incomplete knowledge about the system may specify his intuition of the protocol and get some properties that may not be satisfied by the actual protocol.

2 The PPC$_{\nu\sigma}$ model

The process algebra PPC$_{\nu\sigma}$ extends semantically the *Probabilistic Polynomial-time Process Calculus PPC* [11] to better take into account the analysis of probabilistic security protocols.

2.1 Syntax of PPC$_{\nu\sigma}$

Terms. The set of terms $\mathcal{T}$ of the process algebra PPC$_{\nu\sigma}$ consists of variables $\mathcal{V}$, numbers $\mathbb{N}$, pairs and a specific term $\mathbf{N}$ standing for the security parameter. Security parameter may be understood as cryptographic primitives key length and may appear in the probabilistic polynomial functions defined below. Formally we have

$$t ::= n \ (integer) \mid x \ (variable) \mid \mathbf{N} \ (secur.\ param.) \mid (t,t) \ (pair)$$

For each term t, $fv(t)$ is the set of variables in t. A *message* is a closed term (i.e. not containing variables). The set of messages is denoted $\mathcal{M}$.

Functions. The call of probabilistic as well as deterministic cryptographic primitives, such as keys and nonces generation, encryption, decryption, etc., is modeled by probabilistic polynomial functions [2] $\Lambda : \mathcal{M}^k \to \mathcal{M}$ satisfying

$$\forall (m_1, \ldots, m_k) \in \mathcal{M}^k,\ \forall m \in \mathcal{M},\ \forall \lambda \in \Lambda,\ \exists p \in [0,1] \text{ such that } \mathrm{Prob}[\lambda(m_1, \cdots, m_k) = m] = p.$$

We denote $\lambda(m_1, \cdots, m_k) \hookrightarrow x$ the assignment of the value $\lambda(m_1, \cdots, m_k)$ to the variable x and by $\lambda(m_1, \cdots, m_k) \overset{p}{\hookrightarrow} m$ if $\lambda(m_1, \cdots, m_k)$ evaluates to m with probability p. From Definition A.1 (see Appendix A), the set

$$\mathrm{Im}(\lambda(m_1, \cdots, m_k)) = \{m \mid \exists p \in]0..1]\ \lambda(m_1, \cdots, m_k) \overset{p}{\hookrightarrow} m\}$$

of m s.t. $\lambda(m_1, \cdots, m_k)$ evaluates to m with non-zero probability is finite.

[2] See Appendix A for a formal definition of a *probabilistic polynomial function*

For instance, RSA encryption which takes as parameters, a message m to encrypt and an encryption key formed of the pair (e, n), and returns the number $m^e \ mod \ n$, is the function $\lambda_{RSA}(m, e, n)$ returning c with probability 1 if $c = m^e \ mod \ n$, and 0 otherwise. Similarly, we can model the key guessing attack of a cryptosystem by the product $[\text{rand}(1^k) \hookrightarrow \ key][\text{dec}(c, key) \hookrightarrow \ x]$ where 1^k is the size of the key randomly generated by the function rand, and the decryption function dec returns m with probability 1 if c is the cryptogram of m encrypted by k and $key = k$. The success of such an attack has probability $p = \frac{1}{2^{|k|}}$. These few examples illustrate the expressive power offered by these functions. We limit ourselves to the probabilistic polynomial ones in order to model all attacks realizable (in the model) in polynomial time.

Processes. Let $\mathcal{C}$ be a countable set of public channels. We assume that each channel is equipped with a bandwidth given by the polynomial function $bw : \mathcal{C} \longrightarrow \mathbb{N}$. We say that a message m belongs to the domain of a channel c, written $m \in \text{dom}(c)$, if the message length $|m|$ is less than or equal to the channel bandwidth, i.e. $m \in \text{dom}(c) \iff |m| \leq bw(c)$. Note that $|(m, m')| = |m| + |m'| + r$ where r is the length of a fixed bits string that allowed us to concatenate and decompose two terms without any ambiguity.

Processes in $\text{PPC}_{\nu\sigma}$ are built as follows :

$$\begin{aligned} P ::= \mathbf{0} \quad &| \quad c(x).P \quad | \quad \overline{c}(m).P \quad | \quad P|P \quad | \quad (\nu c)P \quad | \quad [t = t]P \quad | \\ &| \quad !_{q(\mathbf{N})}P \quad | \quad [\lambda(t_1, \cdots, t_n) \hookrightarrow x]P \end{aligned}$$

Given a process P, the set $fv(P)$ of *free variables*, is the set of variables x in P which are not in the scope of any prefix either input (of the form $c(x)$) or probabilistic evaluation (of the form $[\lambda(t_1, \cdots, t_n) \hookrightarrow x]$). A process without free variables is called *closed* and the set of closed processes is denoted by $\mathcal{P}roc$. Hereafter, all processes are considered closed.

The mechanisms for reading, emitting, parallel composition, restriction and matching are standard. The finite replication $!_{q(\mathbf{N})}P$ is the $q(\mathbf{N})-$fold parallel composition of P with itself, where q is a polynomial function. The novelty is the call and the return of probabilistic polynomial functions

$$[\lambda(t_1, \cdots, t_n) \hookrightarrow x]P$$

This feature allows to model (probabilistic) polynomial cryptographic primitives as well as the probabilistic character of a protocol. Actually, it is the main source of probability in this model.

2.2 Operational semantics

The set of actions

$$\mathcal{A}ct = \{\overline{c}(m), c(m), \overline{c}(m) \cdot c(m), c(m) \cdot \overline{c}(m), \tau \mid m \in \mathcal{M} \text{ and } c \in \mathcal{C}\}$$

consists of the set of partial input and output actions, of the set of synchronization actions on public channels, and of the internal action τ:

$$\begin{aligned} \mathcal{P}artial &= \{\overline{c}(m), c(m) \mid m \in \mathcal{M} \text{ and } c \in \mathcal{C}\} \\ \mathcal{A}ctual &= \{\overline{c}(m) \cdot c(m), c(m) \cdot \overline{c}(m), \tau \mid m \in \mathcal{M} \text{ and } c \in \mathcal{C}\} \end{aligned}$$

The set of observable actions is given by $\mathcal{V}is = \mathcal{A}ct - \{\tau\}$. The operational semantics of PPC$_{\nu\sigma}$ is a probabilistic transition system $(\mathcal{E}, \mathcal{T}, E_0)$ generated by the inference rules given in Table 1 where $\mathcal{E} \subseteq \mathcal{P}roc$ is the set of states, $\mathcal{T} \subseteq \mathcal{E} \times Act \times [0,1] \times \mathcal{E}$ the set of transitions and $E_0 \in \mathcal{P}roc$ the initial state. The notation $P \xrightarrow{\alpha[p]} P'$ stands for $(P, \alpha, p, P') \in \mathcal{T}$. It is an extension of the *CCS* semantics, with a mechanism for calling probabilistic polynomial functions. We sketch it briefly here.

To make sure that internal computations of functions do not interfere with communication actions and in particular with those on public channels controlled by the intruder, all exposed functions in a process (`Eval` rule) are simultaneously evaluated by the probabilistic polynomial function *eval* defined in Table 2 below. This evaluation step allows to get what we call a *blocked process* that is, a process having no more internal computations to perform. The set of blocked processes is denoted by $\mathcal{B}locked$. The output mechanism allows a principal A to send a message on public channels (`Output` rule). Dually, the input mechanism must be ready to receive any message on a public channel (`Input` rule). The parallelism (`Par.` rules) operator is defined as usual. It is worth noting that the semantics keep track of information involved into an interaction (the message and the communication channel) (`Syn.` rules), contrarily to most of process algebra semantics where this information is lost as the only action resulting from such a communication is usually the invisible action τ. The restriction operator ν is used to model private channels. The process $(\nu c)P$ behaves like P restricted to actions not on c unless a synchronization occurs on c (i.e. actions of the form $c(m) \cdot \overline{c}(m)$ or $\overline{c}(m) \cdot c(m)$). In this case, they are observed as an invisible action τ (`Rest.` rules).

Note that transitions systems generated by the operational semantics (Table 1) of PPC$_{\nu\sigma}$ processes are not purely probabilistic. Consider for example process $P = \overline{c_1}(a)|\overline{c_2}(b)$. Clearly the sum of probabilities of outgoing transitions of P is equal to 2. It is due to the parallel composition which introduces nondeterminism. In order to resolve this nondeterminism it is mandatory to schedule at each evaluation step of the process, all available distinct actions. However, security protocols

$$\texttt{Eval.}\ \frac{\mathit{eval}(P) \overset{p}{\hookrightarrow} P' \quad P \notin \mathcal{B}locked}{P \xrightarrow{\tau[p]} P'}$$

$$\texttt{Output}\ \frac{m \in \mathrm{dom}(c)}{\overline{c}(m).P \xrightarrow{\overline{c}(m)[1]} P} \qquad \texttt{Input}\ \frac{m \in \mathrm{dom}(c)}{c(x).P \xrightarrow{c(m)[1]} P[m/x]}$$

$$\texttt{ParL.}\ \frac{P_1 \xrightarrow{\alpha[p]} P_1' \quad (P_1|P_2) \in \mathcal{B}locked}{P_1|P_2 \xrightarrow{\alpha[p]} P_1'|P_2} \qquad \texttt{ParR.}\ \frac{P_2 \xrightarrow{\alpha[p]} P_2' \quad (P_1|P_2) \in \mathcal{B}locked}{P_1|P_2 \xrightarrow{\alpha[p]} P_1|P_2'}$$

$$\texttt{SyncL.}\ \frac{P_1 \xrightarrow{\overline{c}(m)[p_1]} P_1' \quad P_2 \xrightarrow{c(m)[p_2]} P_2'}{P_1|P_2 \xrightarrow{\overline{c}(m).c(m)[p_1 . p_2]} P_1'|P_2'} \qquad \texttt{SyncR.}\ \frac{P_1 \xrightarrow{c(m)[p_1]} P_1' \quad P_2 \xrightarrow{\overline{c}(m)[p_2]} P_2'}{P_1|P_2 \xrightarrow{c(m).\overline{c}(m)[p_1 . p_2]} P_1'|P_2'}$$

$$\texttt{RestCL.}\ \frac{P \xrightarrow{\overline{c}(m).c(m)[p]} P'}{(\nu c)P \xrightarrow{\tau[p]} (\nu c)P'} \qquad \texttt{RestCR.}\ \frac{P \xrightarrow{c(m).\overline{c}(m)[p]} P'}{(\nu c)P \xrightarrow{\tau[p]} (\nu c)P'}$$

$$\texttt{Rest}\ \frac{P \xrightarrow{\alpha[p]} P' \quad \alpha \notin \{c(m), \overline{c}(m), \overline{c}(m) \cdot c(m), c(m) \cdot \overline{c}(m) : m \in \mathcal{M}\} \quad P \in \mathcal{B}locked}{(\nu c)P \xrightarrow{\alpha[p]} (\nu c)P'}$$

Table 1
Operational semantics of PPC$_{\nu\sigma}$

$$\begin{array}{l}
\text{Prob}[\mathit{eval}(\mathbf{0}) = \mathbf{0}] = 1 \\
\text{Prob}[\mathit{eval}(c(x).P) = c(x).P] = 1 \ \text{ and } \ \text{Prob}[\mathit{eval}(\overline{c}(m).P) = \overline{c}(m).P] = 1 \\
\text{Prob}[\mathit{eval}((\nu c)P) = (\nu c)Q] = \text{Prob}[\mathit{eval}(P) = Q] \\
\begin{cases} \text{Prob}[\mathit{eval}([m = m']P) = Q] = \text{Prob}[\mathit{eval}(P) = Q] & \text{if } m = m' \\ \text{Prob}[\mathit{eval}([m = m']P) = \mathbf{0}] = 1 & \text{else} \end{cases} \\
\text{Prob}[\mathit{eval}(P|Q) = P'|Q'] = \text{Prob}[\mathit{eval}(P) = P'] \times \text{Prob}[\mathit{eval}(Q) = Q'] \\
\text{Prob}[\mathit{eval}([\lambda(m_1, \cdots, m_k) \hookrightarrow x]P) = Q] = \\
\qquad \sum_{m \in Im(\lambda(m_1, \cdots, m_k))} \text{Prob}[\lambda(m_1, \cdots, m_k) = m] \times \text{Prob}[\mathit{eval}(P[m/x]) = Q]
\end{array}$$

Table 2
Reduction of unblocked processes

are assumed to be executed in hostile environments, that is, with external intruders having full control of the communication network, and that may assign any probabilistic distribution to the controlled channels, i.e. to the public actions. This is modeled by putting public actions under control of an external scheduler. Since we don't want to define a particular attack strategy, the scheduling is not included in the semantics of processes, but rather in the intruder's definition: the intruder is then formed by the pair (Π, S) of process Π and scheduler S. One may view the hostile environment as Π interacting with the protocol, and S as its attack strategy.

2.3 External Scheduler

Given a protocol P attacked by the intruder (Π, S), evaluation of $P|\Pi$ along the strategy S is a four step process consisting of:

Reduction: evaluation of all exposed probabilistic functions in $P|\Pi$.

Localization: indexing of executable actions along $\mathit{eval}(P|\Pi)$ to discriminate whether an executable action interferes or not with an intruder's action.

Selection: scheduling [3] S among available actions.

Execution: the action chosen by S is executed and process is repeated until there is no more executable action.

Localization. Scheduling should discriminate whether the intruder is attached to an action or not. Since a system P attacked by the intruder Π is simply modeled by $P|\Pi$, actions are indexed by the positions of the components they belong to, e.g. if P and Π have respectively n and k parallel components, then $P|\Pi$ consists of $n+k$ components. By convention the attacker is on the right side, so that actions indexed by integers less than or equal to n belong to the protocol while those indexed by integers greater than n belong to the intruder. A partial action is indexed with an integer denoting the component to which it belongs, while a communication action is indexed by a pair of integers denoting the components to which complementary partial actions belong.

Let $\mathcal{I}ndex = \mathbb{N} \cup \mathbb{N}^2$ and the function support : $Act \setminus \{\tau\} \to \mathcal{C}$ where support(α) is the channel name where α occurred.

Definition 2.1 The localization function χ : $\mathcal{B}locked \longrightarrow 2^{\mathcal{A}ct \times \mathcal{I}ndex}$ is defined recursively as follows:

[3] Note that scheduling is defined only for blocked processes. Indeed, the only action available to an unblocked process is the internal action corresponding to functions evaluation.

$$\begin{aligned}
\chi(\mathbf{0}) &= \emptyset \\
\chi(\alpha.P) &= \{(\alpha, 1)\} \\
\chi(P|Q) &= \chi(P) \cup \{(\alpha, \rho(P) + i) \mid (\alpha, i) \in \chi(Q)\} \\
&\quad \cup \{(\alpha \cdot \overline{\alpha}, i, \rho(P) + j) \mid (\alpha, i) \in \chi(Q) \text{ and } (\overline{\alpha}, j) \in \chi(Q)\} \\
\chi((\nu c)P) &= \{(\tau, i, j) \mid (\alpha \cdot \overline{\alpha}, i, j) \in \chi(P) \text{ and } \mathrm{support}(\alpha) = c\}
\end{aligned}$$

where $\alpha \in \mathcal{P}artial$, $\max(i, (j, k)) = \max(i, j, k)$. and

$$\rho(P) = \begin{cases} \max\{ID \in \mathcal{I}ndex \mid (\beta, ID) \in \chi(P),\ \beta \in \mathcal{V}is\} & \text{if } \chi(P) \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

Selection. The function χ allows to localize actions, but for knowing whether an indexed action interferes or not with an intruder's component, more guidelines are needed. Actually, here is needed a partition of the set of indexed actions into *classes of strategically equiprobable actions*, that is, classes of actions uniformly chosen in a strategy S. Intuitively, a class corresponds to actions which can not be discriminated by any scheduler. The construction of the quotient set must agree with the following principles:

(1) No strategy discriminates internal actions of a protocol.

(2) No strategy allows the intruder to control internal reactions of a protocol P to any external stimulus. So, if P can react in many positions to a stimulus of the intruder, then all these positions must have the same probability to react to the intruder's request.

(3) In any strategy, the intruder has complete control on its own actions.

Given a protocol P and an attacker Π, we use χ to compute two sets I_1 and I_2 s.t. indices corresponding to the protocol's components belong to I_1 and those corresponding to the intruder's ones belong to I_2. Formally:

$$I_1 = \begin{cases} \{1, 2, \cdots, \rho(\mathit{eval}(P))\} & \text{if } \chi(\mathit{eval}(P)) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$I_2 = \begin{cases} \{\rho(\mathit{eval}(P)) + 1, \cdots, \rho(\mathit{eval}(P)) + \rho(\mathit{eval}(\Pi))\} & \text{if } \chi(\mathit{eval}(\Pi)) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

The quotient set of strategically equiprobable actions is summarized in Table 3 where $[(\alpha, ID)]_{I_1 \times I_2}$ denotes the equiprobable class of the indexed action (α, ID) w.r.t. sets I_1 and I_2. Let us describe briefly these classes.

Due to principle (1), *internal actions* of P are equiprobable: it is reflected in the definition of $[(\tau, i, j)]_{I_1 \times I_2}$; τ actions indexed by the components positions of P (i.e. in I_1) are equivalent. Otherwise $[(\tau, i, j)]_{I_1 \times I_2}$ is reduced to itself w.r.t (3). Due to principle (2), *partial outputs* on a given public channel are equiprobable: although the intruder can choose the public channel to spy, he has no control on messages transmitted on it. Otherwise $[(\overline{c}(m), i)]$ is reduced to itself: being an intruder's action, he can choose both the message and the component to build his attack. *Partial input* is the dual case of partial output, but contrarily to it, the intruder controls the message (sent by himself) received by P. The same principles apply to *public synchronization*. The intruder can choose a listening channel c and act just as an observer, then any communication on c takes place between two components of P. He has no control either on the message exchanged or on components where communication occurs. However, if it arises from the protocol (output) to the

$$
\begin{array}{lll}
[(\tau,i,j)]_{I_1\times I_2} & = & \begin{cases} \{(\tau,i',j') \mid i',j' \in I_1\} & \text{if } i,j \in I_1 \\ \{(\tau,i,j)\} & \text{otherwise} \end{cases} \\
[(\overline{c}(m),i)]_{I_1\times I_2} & = & \begin{cases} \{(\overline{c}(m'),i') \mid i' \in I_1, m' \in \mathrm{dom}(c)\} & \text{if } i \in I_1 \\ \{(\overline{c}(m),i)\} & \text{otherwise} \end{cases} \\
[(c(m),i)]_{I_1\times I_2} & = & \begin{cases} \{(c(m),j) \mid j \in I_1\} & \text{if } i \in I_1 \\ \{(c(m),i)\} & \text{otherwise} \end{cases}
\end{array}
$$

$$
[(\overline{c}(m)c(m),i,j)]_{I_1\times I_2} = \begin{cases} \{(\alpha\overline{\alpha},i',j') \mid i',j' \in I_1, \mathrm{support}(\alpha)=c\} & \text{if } i,j \in I_1 \\ \{(\overline{c}(m')c(m'),i',j) \mid i' \in I_1, m' \in \mathrm{dom}(c)\} & \text{if } i \in I_1, j \in I_2 \\ \{(\overline{c}(m)c(m),i,j)\} & \text{otherwise} \end{cases}
$$

$$
[(c(m)\overline{c}(m),i,j)]_{I_1\times I_2} = \begin{cases} \{(\alpha\overline{\alpha},i',j') \mid i',j' \in I_1, \mathrm{support}(\alpha)=c\} & \text{if } i,j \in I_1 \\ \{(c(m)\overline{c}(m),i',j) \mid i' \in I_1\} & \text{if } i \in I_1, j \in I_2 \\ \{(c(m)\overline{c}(m),i,j)\} & \text{otherwise} \end{cases}
$$

Table 3
Strategically equiprobable actions

intruder (input), then the intruder can fix the channel and his (input) component. For synchronization arising in the opposite direction, the intruder can fix not only the channel but also the message and his output component as well. Finally, if it arises between two intruder's components then he controls everything.

Definition 2.2 [External scheduler] An *external scheduler* is a stochastic polynomial probabilistic function $S : 2^{Act\times\mathcal{I}ndex} \times 2^{\mathbb{N}} \times 2^{\mathbb{N}} \to Act \times \mathcal{I}ndex$ s.t. for any non empty set $A \subseteq Act \times \mathcal{I}ndex$ and any pair of sets $I_1, I_2 \subseteq \mathbb{N}$ (with $I_1 \neq \emptyset$ or $I_2 \neq \emptyset$) satisfying $\forall_{(i_1,i_2)\in I_1\times I_2} i_1 < i_2$, the following holds:

(i) $\sum_{(\tau,i,j)\in A, i,j\in I_1} \mathrm{Prob}[S(A,I_1,I_2) = (\tau,i,j)] \in \{0,1\}$.

(ii) $\forall_{\alpha,\beta\in A}\alpha \in [\beta]_{I_1\times I_2} \Rightarrow \mathrm{Prob}[S(A,I_1,I_2)=\alpha] = \mathrm{Prob}[S(A,I_1,I_2)=\beta]$.

The set of schedulers is denoted by *Sched*[4].

From the stochasticity condition, being itself a progress condition since it states that at each step of the process at least one of the executable actions will be scheduled, we have the following result.

Lemma 2.3 *Let P be a process s.t. $A = \chi(eval(P)) \neq \emptyset$ then the following holds:* $\forall_{S\in Sched}\exists_{\alpha\in A}\mathrm{Prob}[S(A)=\alpha] \neq 0$.

Our main result on schedulers follows.

Theorem 2.4 *The sum of probabilities of outgoing transitions in any state along any external scheduler is smaller than or equal to 1.*

2.4 Cumulative probability distribution

Transitions systems induced by the operational semantics of Table 1 may have a state P with several outgoing transitions labeled by the same action and the same probability. But to compute correctly the probability of outgoing transitions of P

[4] Given a protocol P and an intruder (Π, S), we know how to compute I_1 et I_2 induced from $A = \chi(P|\Pi)$. Hereafter, by sake of simplicity, we implicitely write $S(A)$ for $S(A, I_1, I_2)$.

according to a scheduler, we must ensure ourself that they can be uniquely identifiable. If P is blocked then χ enables us to uniquely index the outgoing transitions of P. If P is unblocked then there exists a finite number $n = |Im(eval(P))|$ of processes Q_i $(1 \leq i \leq n)$ s.t. $\exists_{q_i \neq 0}\ P \xrightarrow{\tau[q_i]} Q_i$ is an outgoing transition of P. We can order Q_i from 1 to n and use this ordering to index outgoing transitions of P s.t. $P \xrightarrow{\tau[q_i]} Q_i$ being indexed by (i, i) [5].

Let $\sigma = (\alpha_1, id_1) \dots (\alpha_n, id_n)$ be a sequence of indexed actions. Then σ is a path from P to Q if there exists nonzero probabilities $p_1, \dots, p_n$ s.t.

$$P_0 \xrightarrow{\alpha_1[p_1]} P_1 \xrightarrow{\alpha_2[p_2]} \cdots \xrightarrow{\alpha_n[p_n]} P_n,$$

$P = P_0$ and $Q = P_n$. Similarly, we say that P reaches Q by path σ with probability p according to S, denoted by $P \overset{\sigma[p]}{\Longrightarrow}_S Q$, if the probability that S chooses σ is $\text{Prob}[S(\sigma)] = \prod_{1 \leq i \leq n} q_i = p$ where

$$q_i = \begin{cases} p_i & \text{if } P_{i-1} \notin \mathcal{B}locked \\ \text{Prob}[S(\chi(P_{i-1})) = (\alpha_i, id_i)] \times p_i & \text{otherwise} \end{cases}$$

An α-*path* is a path of type $(\tau, id_1)(\tau, id_2) \dots (\tau, id_{n-1})(\alpha, id_n)$ $(n \geq 1)$. The notation $P \overset{\alpha[p]}{\Longrightarrow}_S Q$ means that there exists an α-path σ s.t. $P \overset{\sigma[p]}{\Longrightarrow}_S Q$. Similarly $P \overset{\hat{\alpha}[p]}{\Longrightarrow}_S Q$ denotes $P \overset{\alpha[p]}{\Longrightarrow}_S Q$ if $\alpha \neq \tau$ and $P \overset{\tau^*[p]}{\Longrightarrow}_S Q$ otherwise.

Let $\mathscr{E} \subseteq \mathcal{P}roc$ be a set of processes and $Q \in \mathscr{E}$. Let P be a process, and σ an α-path from P to Q. Then σ is *minimal* w.r.t $\mathscr{E}$ if no other α-path σ' exists from P to Q' s.t. σ' is a prefix [6] of σ and $Q' \in \mathscr{E}$. We denote by $Paths(P, \overset{\alpha}{\Longrightarrow}, \mathscr{E})$ the set of all minimal α-paths from P to an element of $\mathscr{E}$.

Definition 2.5 Let $\mathscr{E} \subseteq \mathcal{P}roc$ be a set of processes. The total probability that P reaches a process in $\mathscr{E}$ by an α-path according to S is computed by the *cumulative probability function* $\mu : \mathcal{P}roc \times Act \times 2^{\mathcal{P}roc} \times Sched \to [0, 1]$

$$\mu(P, \overset{\hat{\alpha}}{\Longrightarrow}_S, \mathscr{E}) = \begin{cases} 1 & \text{if } P \in \mathscr{E}, \alpha = \tau \\ \sum\{\text{Prob}[S(\sigma)] : \sigma \in Paths(P, \overset{\alpha}{\Longrightarrow}, \mathscr{E})\} & \text{otherwise} \end{cases}$$

Theorem 2.6 *The cumulative probability function is well defined i.e.* $\forall_{P,\alpha,\mathscr{E},S}\ \mu(P, \overset{\hat{\alpha}}{\Longrightarrow}_S, \mathscr{E}) \leq 1$.

3 Probabilistic behavioural equivalences

Now we plan to establish equivalences ensuring that a protocol satisfies a security property if and only if it is observationally equivalent to an abstraction of the protocol, satisfying the security property by construction. In other words we request two processes to be equivalent if and only if, when subject to same attacks, they generate "approximately" the same observations. By "approximately" we mean *asymptotically* closed w.r.t. the security parameter [7].

[5] by analogy to the indexing of τ actions by χ

[6] Note that prefixing does note care about index, i.e. (α_1, id_3) is a prefix of $(\alpha_1, id_1)(\alpha_2, id_2)$ for all index id_1 and id_3. Note also that the minimality condition applies only to τ-paths.

[7] For verification purpose, all along this section, we consider only actual actions (i.e. actions in $\mathcal{A}ctual$) keeping in mind that partial actions are never executable. So far partial actions have been considered purely

3.1 Asymptotic observational equivalence

We start by defining the notion of an observable and the probability that a given process P generates a particular observable. An observable is simply a pair (c, m) of a public channel and a message. The set of all observables is denoted by $\mathcal{O}bs$. The probability of observing (c, m) is defined as the sum of the probability of observing it directly, i.e. the cumulative probability of executing an $\overline{c}(m) \cdot c(m)$-path or an $c(m) \cdot \overline{c}(m)$-path to reach any state, and the probability of observing it indirectly, i.e. by observing first some different visible actions before observing it. For that purpose we extend the notion of cumulative probability to the so-called *cumulative probability up to* H.

Definition 3.1 Let $\mathscr{E} \subseteq \mathcal{P}roc$ be a set of processes, P a process, S a scheduler and $H \subset \mathcal{A}ctual \setminus \{\tau\}$ a set of visible actions. The *cumulative probability up to* H is defined inductively as follows: $\forall \alpha \in \mathcal{A}ctual \setminus H$

$$\mu(P, \stackrel{\hat{\alpha}}{\Longrightarrow}_{S/H}, \mathscr{E}) = \begin{cases} 1 \quad \text{if } P \in \mathscr{E} \text{and } \alpha = \tau, \\ \mu(P, \stackrel{\hat{\alpha}}{\Longrightarrow}_{S}, \mathscr{E}) + \\ \sum_{\beta \in H,\ Q \in \mathcal{P}roc} \mu(P, \stackrel{\hat{\beta}}{\Longrightarrow}_{S}, \{Q\}) \mu(Q, \stackrel{\hat{\alpha}}{\Longrightarrow}_{S/H}, \mathscr{E}) \ \text{ otherwise} \end{cases}$$

Lemma 3.2 *The cumulative probability up to* H *is well defined, i.e.* $\forall P, \alpha, \mathscr{E}$, S *and* H, $\mu(P, \stackrel{\hat{\alpha}}{\Longrightarrow}_{S/H}, \mathscr{E}) \leq 1$.

Definition 3.3 Let $o = (c, m)$ be an observable and $L_o = \{\overline{c}(m) \cdot c(m),\ c(m) \cdot \overline{c}(m)\}$. The cumulative probability that P generates o according to S is

$$\text{Prob}[P \rightsquigarrow_S o] = \sum_{\alpha \in L_o} \mu(P, \stackrel{\hat{\alpha}}{\Longrightarrow}_{S/(\mathcal{A}ctual \setminus (L_o \cup \{\tau\}))}, \mathcal{P}roc).$$

Lemma 3.4 *The cumulative probability of an observable is well defined, i.e.* $\forall P, o$, *and* S, $\text{Prob}[P \rightsquigarrow_S o] \leq 1$.

We define our asymptotic observational equivalence relation stating that two processes are equivalent if they generate the same observables with approximately the same probabilities when they are attacked by the same enemy.

Definition 3.5 Let $\mathcal{P}oly : \mathbb{N} \to \mathbb{R}^+$ be the set of positive polynomials and $\mathcal{E} = \mathcal{P}roc \times \mathcal{S}ched$ the set of attackers. Two processes P and Q are observational equivalent, denoted by $P \simeq Q$, iff $\forall_{q \in \mathcal{P}oly}$, $\forall_{o \in \mathcal{O}bs}$, $\forall_{(\Pi, S) \in \mathcal{E}}$, $\exists_{i_0}$ s.t. $\forall_{\mathbf{N} \geq i_0}$

$$|\text{Prob}[P|\Pi \rightsquigarrow_S o] - \text{Prob}[Q|\Pi \rightsquigarrow_S o]| \leq \frac{1}{q(\mathbf{N})}$$

Theorem 3.6 $\simeq$ *is an equivalence relation.*

In order to develop methods for reasoning about security properties of crypto-protocols based on observable traces, we reformulate the observational equivalence to take into account any observable trace.

---

by sake of semantic soundness and completeness.

3.2 Trace equivalence

We start by defining the cumulative probability that a process P generates a sequence of observables $o_1o_2\cdots o_n$ recursivelly as the probability that it generates directly the first observable o_1 and reach any state Q times the cumulative probability that the process Q generates the remaining sequence.

Definition 3.7 Let $o_1o_2\cdots o_n$ be a sequence of observables s.t. $\forall i \leq n, o_i = (c_i, m_i)$ and P be a process. Let $\alpha_i = \overline{c_i}(m_i)\cdot c_i(m_i)$ and $\beta_i = c_i(m_i)\cdot\overline{c_i}(m_i)$ $\forall 1 \leq i \leq n$. The cumulative probability that P generates the sequence $o_1o_2\cdots o_n$ according to scheduler S is

$$\begin{aligned}&\text{Prob}[\, P \rightsquigarrow_S^{tr} o_1o_2\cdots o_n]\\ &\quad= \sum_{Q\in\mathcal{P}roc}(\mu(P, \stackrel{\hat{\alpha_1}}{\Longrightarrow}_S, \{Q\}) + \mu(P, \stackrel{\hat{\beta_1}}{\Longrightarrow}_S, \{Q\}))\text{Prob}[Q \rightsquigarrow_S^{tr} o_2\cdots o_n].\end{aligned}$$

Lemma 3.8 *The cumulative probability of observing a sequence of observables is well defined, i.e.* $\forall P, o_1, o_2, \cdots, o_n$ *and* S, $\text{Prob}[P \rightsquigarrow_S^{tr} o_1o_2\cdots o_n] \leq 1$.

Definition 3.9 Two processes P and Q are trace equivalent, denoted by $P \simeq^{tr} Q$, iff $\forall_{q\in\mathcal{P}oly}$, $\forall_{o_1,o_2,\cdots,o_n\in\mathcal{O}bs}$, $\forall_{(\Pi,S)\in\mathcal{E}}$, $\exists_{i_0}$ s.t. $\forall_{\mathbf{N}\geq i_0}$

$$|\text{Prob}[P|\Pi \rightsquigarrow_S^{tr} o_1o_2\cdots o_n] - \text{Prob}[Q|\Pi \rightsquigarrow_S^{tr} o_1o_2\cdots o_n]| \leq \frac{1}{q(\mathbf{N})}$$

Theorem 3.10 $\simeq^{tr}$ *is an equivalence relation.*

Theorem 3.11 *Trace equivalence is equivalent to the observational equivalence, i.e.*

$$\forall_{P,Q\in\mathcal{P}roc} P \simeq^{tr} Q \Leftrightarrow P \simeq Q.$$

4 Case study: the Dining Cryptographers protocol

The *Dining Cryptographers* [8] protocol is a paradigmatic example of a protocol which ensures anonymity property. Its author defines it as follows:

> *Three cryptographers are sitting down to dinner at their favorite restaurant. Their waiter informs them that arrangements have been made with the maitre d'hotel for the bill to be paid anonymously. One of the cryptographers might be paying for the dinner, or it might have been NSA (U.S. National Security Agency). The three cryptographers respect each other's right to make an anonymous payment, but they wonder if NSA is paying.*

To fairly resolve their uncertainty, they carry out the following protocol: each cryptographer flips a coin between him and the one on his right, so that only the two of them can see the outcome. Each one then states whether the two coins he can see fell on the same side or on different sides. A payer (if any!) states the opposite of what he sees. The idea is that if the coins are unbiased and the protocol is carried out faithfully then an odd number of"different" indicates that one of them is paying and neither of the other two learns anything about his identity; otherwise NSA is paying.

4.1 A flawed specification of the protocol

In the following specification [8] we suppose that the NSA makes his choice according to a probabilistic distribution (known only by him-self) defined by the function λ_{NSA} and informs each cryptographer over a secure channel if he is the payer or not. To ensure fairness beetwen cryptographers, that is no one having advantage over another, each coin flipping is made by an "outside trusted third party" thanks to the function $flips$ and the result made avalable to both concerned cryptographers.

$$
\begin{array}{lll}
NSA ::= & [\lambda_{NSA}(3) \hookrightarrow x](\prod_{0\leq i\leq 3}[x=i]Payer_i) \\
Payer_3 ::= & \overline{c}_0(nopay).\overline{c}_1(nopay).\overline{c}_2(nopay).\mathbf{0} \\
Payer_i ::= & \overline{c}_i(pay).\overline{c}_{i\oplus 1}(nopay).\overline{c}_{i\oplus 2}(nopay).\mathbf{0} \text{ if } 0\leq i\leq 2. \\
Crypts ::= & [flips(coin_0)\hookrightarrow y_0][flips(coin_1)\hookrightarrow y_1][flips(coin_2)\hookrightarrow y_2]\prod_{0\leq i\leq 2} Crypt_i \\
Crypt_i ::= & c_i(z_i).([z_i = pay]P_i|[z_i = nopay]Q_i) \\
P_i ::= & [y_i = y_{i\oplus 1}]\overline{pub}_i(desagree).\mathbf{0}|[y_i \neq y_{i\oplus 1}]\overline{pub}_i(agree).\mathbf{0} \\
Q_i ::= & [y_i = y_{i\oplus 1}]\overline{pub}_i(agree).\mathbf{0}|[y_i \neq y_{i\oplus 1}]\overline{pub}_i(desagree).\mathbf{0}
\end{array}
$$

The protocol is then specified as follows: $DC^1 ::= \nu c_0 c_1 c_2(NSA|Crypts)$

4.2 Specification of anonymity

We give a probabilistic version of anonymity specification due to [12] in the possibilistic model CSP. The idea is that given two sets A and O of anonymous and observable events respectively, a protocol P ensures anonymity of events A to any observer who can see only events O if P doesn't allow him to determine any causal dependency beetwen the probabilistic distributions of A and O.

Definition 4.1 [Anonymity property] Let A and O be the sets of anonymous and observable events respectively, Perm(A) is the set of permutations of the elements of A. Let P be a process and $\pi \in \text{Perm}(A)$ be a permutation, we denote by P_π the process obtained by replacing any occurrence of the event a in P by the event $\pi(a)$. Then P ensures anonymity of events A iff

$$\forall_{\pi\in Perm(A)} P \simeq^{tr} P_\pi$$

In the above specification the anonymous events are $A_{DC} = \{(c_i, m) \mid i = 0, 1, 2 \text{ and } m = pay,\ nopay\}$ and the observable events are any communication over a public channel. Let

$$Sched_\tau = \{S \in Sched \mid \sum_{(\tau,i,j)\in A, i,j\in I_1} \text{Prob}[S(A, I_1, I_2) = (\tau, i, j)] = 1\}$$

denote the subset of schedulers that give priority to internal actions of the protocol and $\simeq^{tr}_\tau$ the observational trace equivalence induce by $Sched_\tau$. Then we have the following results which show that $Sched$ can detect the flaw in DC^1 while $Sched_\tau$ cannot.

Theorem 4.2 *With the notation above, the following conditions hold:*

- $\forall_{\pi\in Perm(A_{DC})}$, *if* $\forall_{i=0,1,2}$ Prob$[flips(coin_i) = Head] = \frac{1}{2}$ *then* $DC^1 \simeq^{tr}_\tau DC^1_\pi$.

[8] where $\oplus$ is the addition modulus 2.

- *Whatever the probabilistic distributions of the coins are, if π is not the identity permutation then $DC^1 \not\approx^{tr} DC^1_\pi$.*

The flaw in DC^1 results from the fact that the real payer (if any) has advantage over the others since he always gets his message before them. A scheduler that give priority to observable actions (i.e. an intruder who attack the protocol as soon as possible) will generate only observable traces begining only by the real payer public channel. Under such schedulers, DC^1 and DC^1_π will generate different observable traces and hence are not equivalent.

Acknowledgements. The authors are grateful to the reviewers for their careful reading and helpful comments that improved paper's readability.

References

[1] A. Aldini, M. Bravetti, and R. Gorrieri. A process algebra approach for the analysis of probabilistic non-interference. Technical report, 2002.

[2] M.J. Atallah. *Algorithms and Theory of Computation Handbook*. CRC Press LLC, 1999.

[3] J. Bengt, K.G Larson, and W. Yi. Probabilistic extension of process algebra. In *Handbook of process algebra*, page 565, 2002.

[4] Bruno Blanchet. A computationally sound mechanized prover for security protocols. In *S&P*, pages 140–154. IEEE Computer Society, 2006.

[5] Ran Canetti, Ling Cheung, Dilsun Kirli Kaynar, Moses Liskov, Nancy A. Lynch, Olivier Pereira, and Roberto Segala. Time-bounded task-pioas: A framework for analyzing security protocols. In Shlomi Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 238–253. Springer, 2006.

[6] K. Chatzikokolakis and C. Palamidessi. A Framework for Analysing Probabilistic Protocols and its Applications to the Partial Secrets Exchange. In *Proc. of the Sym. on Trust. Glob. Comp. (STGC'05)*, LNCS. Spr.-Ver., 2005.

[7] K. Chatzikokolakis and C. Palamidessi. Making random choices invisible to the scheduler. In *Proc. of CONCUR'07). To appear.*, 2007.

[8] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptology*, 1(1):65–75, 1988.

[9] Flavio D. Garcia, Peter van Rossum, and Ana Sokolova. Probabilistic anonymity and admissible schedulers. *http://arxiv.org/abs/0706.1019*, 2007.

[10] Peeter Laud and Varmo Vene. A type system for computationally secure information flow. In Maciej Liskiewicz and Rüdiger Reischuk, editors, *FCT*, volume 3623 of *Lecture Notes in Comp. Sc.*, pages 365–377. Springer, 2005.

[11] J.C. Mitchell, A. Ramanathan, A. Scedrov, and V. Teague. A probabilistic polynomial-time process calculus for the analysis of cryptographic protocols. *Theoretical Computer Science*, 353:118–164, 2006.

[12] S. Schneider and A. Sidiropoulos. Csp and anonymity. In *Proc. Comp. Security - ESORICS 96*, volume 1146 of *LNCS*, pages 198–218. Springer-Vale, 1996.

A Probabilistic polynomial functions

The following definitions are standard: see for instance [2] (chapter 24, pp. 19-28).

Definition A.1 A probabilistic function F from X to Y is a function $X \times Y \rightarrow [0,1]$ that satisfies the following conditions.

- $\forall x \in X : \sum_{y \in Y} F(x,y) \leq 1$
- $\forall x \in X$, the set $\{y | y \in Y, F(x,y) > 0\}$ is finite.

For $x \in X$ and $y \in Y$, we say $F(x)$ evaluates to y with probability p, written $\text{Prob}[F(x) = y] = p$, if $F(x,y) = p$.

Definition A.2 The composition $F = F_1 \circ F_2 : X \times Z \rightarrow [0,1]$ of two probabilistic functions $F_1 : X \times Y \rightarrow [0,1]$ and $F_2 : Y \times Z \rightarrow [0,1]$ is the probabilistic function:

$$\forall x \in X, \forall z \in Z : F(x,z) = \sum_{y \in Y} F_1(x,y) \cdot F_2(y,z).$$

Definition A.3 An oracle Turing machine is a Turing machine with an extra oracle tape and three extra states q_{query}, q_{yes} and q_{no}. When the machine reaches the state q_{query}, control is passed either to the state q_{yes} if the contents of the oracle tape belongs to the oracle set, or to the state q_{no} otherwise.

Given an oracle Turing machine M, $M_\sigma(\overrightarrow{a})$ stands for the result of the application of M to $\overrightarrow{a}$ by using the oracle σ.

Definition A.4 An oracle Turing machine executes in polynomial time if there exists a polynomial $q(\overrightarrow{x})$ such that for all σ, $M_\sigma(\overrightarrow{a})$ halts in time $q(|\overrightarrow{a}|)$, where $\overrightarrow{a} = (a_1, \cdots, a_k)$ and $|\overrightarrow{a}| = |a_1| + \cdots + |a_k|$.

Let M be an oracle Turing machine with execution time bounded by the polynomial $q(\overrightarrow{a})$. since $M(\overrightarrow{a})$ may call an oracle with at most $q(\overrightarrow{a})$ bits, we have a finite set $\mathcal{Q}$ of oracles for which M executes in time bounded by $q(\overrightarrow{a})$.

Definition A.5 We say that an oracle Turing machine is *probabilistic polynomial* and write $\text{Prob}[M(\overrightarrow{a}) = b] = p$ the probability that M applied to $\overrightarrow{a}$ returns b is p, if and only if, by choosing uniformly an oracle σ in the finite set $\mathcal{Q}$, the probability that $M_\sigma(\overrightarrow{a}) = b$ is p.

Definition A.6 A probabilistic function F is said *polynomial* if it is computable by a probabilistic polynomial Turing machine, that is, for all input $\overrightarrow{a}$ and all output b, $\text{Prob}[F(\overrightarrow{a}) = b] = \text{Prob}[M(\overrightarrow{a}) = b]$.

Panel Discussion

Information hiding: state-of-the-art and emerging trends

Sabrina De Capitani di Vimercati [1] Steve Kremer[2]
Pasquale Malacaria [3] Peter Ryan[4] David Sands [5]

Abstract

Information hiding is a field where researchers from different areas of computer science have collaborated to solve common/orthogonal problems by means of different techniques and approaches. Information hiding covers aspects like data secrecy, anonymity, database security, information flow, protocol verification... that nowadays are well-established research fields for thousands of researchers around the world.
In this panel, five experts in the field will present approaches like language-based security, quantitative approaches and access control to problems like database security, protocols for anonymity, quantified/functional information flow and automatic protocol verification. They will present their point of view on the topic by answering to the following questions about their field of expertise

(i) which are the issues that nowadays could be declare 'solved'?

(ii) which are the current research issues and how are they currently approached?

(iii) 3. which are the challenges for a near future?

and by taking questions from the audience.

[1] Univ. Milano
[2] INRIA and ENS Cachan
[3] Queen Mary
[4] Newcastle Univ.
[5] Chalmers Univ.