BETTY WG2 State of the art report

Ilaria Castellani[1], Pierre-Malo Deniélou[2], Mariangiola Dezani-Ciancaglini[3], Silvia Ghilezan[4], Jovanka Pantovic[4], Jorge A. Pérez[5], Peter Thiemann[6], Bernardo Toninho[5,8], and Hugo Torres Vieira[7]

[1]INRIA Sophia Antipolis, France
[2]Royal Holloway, University of London, UK
[3]Dipartimento di Informatica, Università di Torino, Italy
[4]Faculty of Technical Sciences, University of Novi Sad, Serbia
[5]CITI and Departamento de Informática, FCT, Universidade Nova de Lisboa, Portugal
[6]University of Freiburg, Germany
[7]LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
[8]Carnegie Mellon University, USA

September 25, 2013

1 Introduction

Types are one of the most popular programming languages techniques to ensure properties of programs, as they allow to single out programs that are error-free (for a certain class of errors) just by inspecting the source code. Examples of errors excluded by typing range from when an object is not able to handle a method call (message not understood) to when concurrent programs are competing (racing) for some resource in an undesired way (leading to inconsistent state or unexpected behaviours). Security properties have also been targeted by type systems (starting with the pioneering work by Volpano, Smith and Irvine [VIS96]), given the importance of excluding security flaws before deploying the programs. Type-based approaches allow, on the one hand, to focus on disciplined systems and ensure that such trusted systems conform to the prescribed security policies (e.g., access control). On the other hand, it is fundamental to develop mechanisms that protect the safe typed world in a real open network, certifying the trusted parties so as to allow the properties that were proved in the typed model to be conveyed to an open setting.

Although behavioural types for communication-centred systems have been studied for a variety of calculi and languages since the late nineties, prominently *session types* [Hon93, HVK98], the study of security properties for such models is still at an early stage. In particular, session-typed models focus on open-ended systems where loosely coupled parties may synchronise to start a session on a specific (public) service, thereafter interacting on the private (isolated)

channel of the session. In general, one could expect security properties to be easier to enforce within sessions than in an open network, since the behaviour of participants within a session is specified by session types and therefore known to be more disciplined than that of generic processes. Moreover, the interaction within a session is supposed to take place on a private channel, accessible only to its participants. Hence, a trusted session-typed setting allows one to reason about systems where participants communicate on privately held channels, following the prescribed protocols — protocol fidelity.

On the one hand, within this trusted platform one may address more specific security properties regarding, e.g., the information that is communicated or the roles in which communication actions are carried out. On the other hand, once we lift the typed-world assumption in such an open-ended dynamic setting we need to certify our trusted typed systems so that their targeted properties, ranging from protocol fidelity to security properties like access control conformance, can be statically ensured also for systems that are deployed in a real open network.

In the remainder of this document we describe some of the existing verification approaches closely related to behavioural type theory that focus on ensuring security properties. Namely, in Section 2 we refer to works that address the regulation of information exchange with untrusted parties, in particular regulating information flow in the setting of multiparty interaction [CCDC13]. In Section 3 we present works that consider an explicit model of distribution and of structured data and govern both the information exchanged between sites and the local information management (e.g., [DCGP06, DCGPV08]). Section 4 describes approaches that focus on ensuring properties of the values exchanged in communications and how may such properties be trusted by communicating partners [TCP11].

To some extent, most of the approaches mentioned above involve enlarging more foundational settings with security concerns and enriching existing techniques so as to tackle the security issues. In a somewhat distinct direction, we find techniques that focus on guaranteeing that the properties studied over more foundational settings can be conveyed to more realistic models considering open networks, namely by compiling session like descriptions to implementations of cryptographic protocols that ensure honest session participants are protected from external interference [CDF$^+$07, CDF$^+$08], presented in Section 5. In a more pragmatical direction, Section 6 presents developments that allow to reconcile legacy code and changing security policies.

2 Security Types for Communication-Centred Calculi

2.1 Introduction

Although behavioural types have been studied for a variety of calculi and languages since the late nineties, the study of security properties for communication-centred calculi and particularly for session calculi is still at an early stage.

In general, one could expect security properties to be easier to enforce within sessions than in an open network, since the behaviour of participants within a session is specified by session types and therefore known to be more disciplined

Note: These first two paragraphs appear also in the general introduction of Section 1. Decide later, when Sction 1 will be completed, whether they should be left here or not.

than that of generic processes. Moreover, the interaction within a session is supposed to take place on a private channel, accessible only to its participants.

An increasingly relevant security issue, as more and more data and services are placed on the internet, is that of preserving the *confidentiality* of privata data hosted on the "cloud" or manipulated by web services and web applications. Protection of data confidentiality requires two complementary techniques: *access control*, which restricts the access to the original data, allowing only trusted users to read them, and *secure information flow*, which prevents the propagation to untrusted users of legally accessed data, thus ensuring end-to-end confidentiality. Hence, compared to access control, secure information flow may be viewed as additionally restricting the access to transformed data, when these have been computed using sensitive data.

Another important security property, which is often presented as the dual of confidentiality and can similarly be expressed as a combination of access control and secure information flow, is data *integrity*. While confidentiality requires that data should not be released to untrusted destinations, integrity requires that data should not originate from untrusted sources.

Security seen in the broad sense of quality of services can also be assured by *reputation* systems which guide principals in the choice of partners in communication protocols.

2.2 Access control

The paper [LPT07] defines a type system for COWS [PT12], a formalism for specifying and combining services, while modelling their dynamic behaviour. This type system permits one to express and enforce *policies* for regulating the exchange of data among services. More precisely, to implement such policies, programmers can annotate data with sets of participant names characterising the participants authorised to use and exchange them. The typed operational semantics uses these annotations to guarantee that computations proceed according to them.

For example, let us consider a standard buyer-seller-bank protocol, in which:

1. the buyer asks an item to the seller and receives a price from the seller together with a bank account number;

2. either the buyer accepts the price and sends his credit card number to the bank, or he turns down the offer.

In this scenario it is clear that only the buyer and the bank, but not the seller, should have access to the credit card number. Therefore the type system of [LPT07] validates processes implementing the protocol described above, but not a variant in which by mistake the buyer would send the credit card number to the seller.

Kolundzija in [Kol08] enriches the calculus of [BM08] (a variant of SCC [BBC+06]) with security levels for controlling access rights. In the original calculus communications can either follow fixed protocols or be disciplined data flows. In the new calculus processes are *framed* [PSS05] by security levels. A process framed by a level ℓ can exercise rights of security level not exceeding ℓ. Security levels are assigned to service definitions, clients and data. In order

to invoke a service, a client must be endowed with an appropriate clearance, and once the service and client agree on the security level, the data exchanged in the initiated session will not exceed this level. The calculus of [Kol08]comes equipped with a type system that statically ensures these security properties.

In the example of the buyer-seller-bank protocol described above, the protection of the credit card number is assured by assuming for the credit card number a level which is incomparable with the level of the seller, but it is smaller or equal to the levels of the buyer and the bank.

The treatment of access control in [CCDC13] is similar to that of [Kol08] since participants and data have security levels, but it enhances the security by taking advantage of the presence of *delegation*. In fact the starting calculus is the calculus of multi-party sessions with delegation first introduced in [HYC08] (in the variant defined in [BCD$^+$08], where no linearity check is needed since channels are identified by session participants). In this calculus, instead of sending a bank connection to the buyer, the seller delegates to the bank the part of the interaction dealing with the credit card. The type system which assures access control has therefore an explicit type constructor to track delegation, which allows the delegated part of a session type to be marked. A main feature of this calculus and type system is a form of *data declassification*. In the previous example of the buyer-seller-bank, the bank should be allowed to tell the seller whether the credit card number of the client is valid or not, of course without revealing the number itself. Declassification here is tailored to communication protocols, since data can be declassified only when they are exchanged.

The comparison between the three described approaches to access control is not easy, since the underlying calculi offer different interaction patterns, typically the primitive for session killing in [LPT07], the pipeline constructor in [Kol08] and the channel delegation in [CCDC13]. In our view, delegation poses the most interesting problems for data protection, since it allows a transparent change of the participant who owns a given communication channel.

2.3 Secure information flow

The paper [CCDC13], which is the full version of [CCDCR10], considers a calculus for multiparty sessions with delegation, enriched with security levels for both participants and data, and defines a bisimulation-based security property for it, formalising the preservation of data confidentiality. It then proposes a session type system for the calculus, adding secure information flow requirements in the typing rules in order to ensure the security property. This study revealed an interesting interplay between the constraints used in security types and those used in session types to ensure properties like communication fidelity and progress.

In this work, the security notion is based on the observation of messages while they are being exchanged. The observation power depends on the level of the observer. An observer of level ℓ can only see messages of security level lower than or equal to ℓ. For simplicity, we assume here just two security levels $\top$ and $\bot$ and we call "secret" or "high" (respectively, "public" or "low"), a message or I/O communication action whose carried value is of security level $\top$ (respectively, $\bot$).

A typical insecure information flow, also called *information leak*, arises when different high inputs cause different low messages to be exchanged. Another source of information leak is the possible blocking of a high input action, because there is no high message to receive from the environment. For instance, the process (where ?, ! denote input, output, respectively, and we use the standard convention on $^+$,$^-$ for endpoints):

$$s^+?(x^\top).s^+! < 1^\bot >$$

emits the low output "1" only if it first receives the high input from the environment. However, this kind of information leak can be sanitised by rendering inputs and outputs persistent, so that high messages are always offered when needed. Hence the former kind of security flaw seems more fundamental.

A following paper [CCDC11] moves one step further by equipping the above calculus with a monitored semantics, which blocks the execution of processes as soon as they attempt to leak information. The safety property induced by this monitored semantics is shown to strictly imply the security property. Indeed, safety is a property of individual computations while security is a property of the set of computations of a process, which may hold even if some of these computations are unsafe.

Altogether, the work of [CCDC13] and [CCDC11] may be summarised as proposing three increasingly precise means for tracking information leaks in sessions: a syntactic property (typability), a local semantic property (safety) and a global semantic property (security).

We illustrate the differences between typability, safety and security by means of a simple example that should convey the intuition, although it does not completely fit the definitions of typability and safety in [CCDC11]. We convene here that typability requires the absence of any "level drop" from the expression tested by a conditional to a subsequent communication, while safety requires the same condition but only in computations that may actually occur. Instead, in [CCDC11] the "level drop" is forbidden only from inputs to subsequent communications (this is sufficient to imply the absence of level drops also in conditionals, because any conditional whose tested expression is high and open must be preceded by a high input), but an example along these lines would require recursive participants for the overall process to be secure, as argued above, and thus it would be too complicated for the present discussion.

Consider a conditional whose $\top$-level condition is true and whose then branch sends a $\bot$-level data on s^+ while its else branch sends a $\top$-level data on s^+ (whose value does not really matter, since this branch is never taken):

$$\text{IF } \mathsf{true}^\top \text{ THEN } s^+! < 1^\bot > \text{ ELSE } s^+! < 2^\top >$$

This process is secure because it always exhibits the same public behaviour, but it is neither safe nor typable. Consider now a variant of the above process, where the two branches of the conditional are swapped:

$$\text{IF } \mathsf{true}^\top \text{ THEN } s^+! < 2^\top > \text{ ELSE } s^+! < 1^\bot >$$

This process is still not typable, but it is now both safe and secure, since the else branch is never taken and thus the level drop cannot occur.

2.3.1 Discussion

There appears to be an influence of classical session types upon security types, which we did not foresee before undertaking our study. Indeed, one of the main causes of insecure information flow in a concurrency scenario is the possibility of different termination behaviours in the branches of a high conditional (a conditional which tests a secret expression). This may give rise to so-called "termination leaks". In session calculi, there are three possible termination behaviours: proper termination, deadlock and divergence. Then, a termination leak may occur for instance if one branch of a high conditional terminates while the other diverges or deadlocks, assuming successful termination is made explicit by an observable action. Session types help containing this phenomenon, by imposing some uniformity in the termination behaviours of conditional branches: for instance, a terminating branch cannot coexist with a diverging branch. They also prevent local deadlocks (due to communication errors within a session) as well as some global deadlocks, thus limiting the possible sources of abnormal termination.

Since the two branches of a conditional must have the same types for all channels, we cannot for example type the process:

$$\text{IF } x^\top \text{ THEN } s^+!<1^\top> \text{ ELSE } \mu X.s^+!<2^\top>.X$$

which could cause a termination leak. Typing also prevents termination leaks due to bad matchings of data, like in the process:

$$\text{IF } x^\top \text{ THEN } s^+!<1^\top> \text{ ELSE } s^+!<x^\top + 3>$$

where we assume that $x^\top$ is replaced by a boolean value. The typing considered until now does not prevent *global deadlocks* due to bad matchings of protocols in interleaved sessions, like in the process:

$$\begin{array}{l}\text{IF } x^\top \text{ THEN } s^+?(y).r^+?(z).s^+!<1^\top>.r^+<2^\top> \\ \qquad \text{ELSE } s^+?(y).s^+!<1^\top>.r^+?(z).r^+<2^\top> \\ \mid s^-!<3^\top>.s^-?(t).r^-<4^\top>.r^-?(u)\end{array}$$

Here, if the THEN branch is taken the process will deadlock, while if the ELSE branch is taken the interaction will terminate successfully.

These global deadlocks are forbidden by the type systems in [Kob02, Pad13].

2.4 Integrity of communicated data

In the standard user-ATM-Bank example, in response to a deposit request by the user a malicious ATM could send to the bank an amount of money that is different from that communicated by the user, consequently altering the balance obtained from the bank. This change is transparent to the typing, since it does not modify the communication protocol. This means that the following processes, where channels s, r are used respectively for the interaction between the user and the ATM and between the ATM and the bank:

$$\begin{array}{lcl}\text{user} & = & s^+!<\mathsf{userId}>.s^+!<\mathsf{depositAmount}> \\ \text{ATM} & = & s^-?(userId).s^-?(depositAmount). \\ & & r^+!<userId>.r^+!<depositAmount-10> \\ \text{bank} & = & r^-?(userId).r^-?(depositAmount)\end{array}$$

can be typed, since the first message sent by the user has type String and the second message has type Int, as expected by the other participants.

In order to cope with such kind of misbehaviour, in [BCG05] Bonelli et al. incorporate *correspondence assertions* [GJ03] in the theory of session types. Two correspondence assertions can be paired by the keywords BEGIN, END and have values which allow the integrity of the communicated data to be checked (in this example userId and depositAmount). The user and the bank processes with correspondence assertions become:

$$\begin{array}{lcl} \text{user} & = & \text{BEGIN}(\mathsf{userId}, \mathsf{depositAmount}).s^+! < \mathsf{userId} > .s^+! < \mathsf{depositAmount} > \\ \text{bank} & = & r^-?(\mathit{userId}).r^-?(\mathit{depositAmount}).\text{END}(\mathit{userId}, \mathit{depositAmount}) \end{array}$$

thus allowing the cheating ATM to be discovered, since the operational semantics requires the same values in paired correspondence assertions. More precisely the reduction rules memorise the values of userId and depositAmount corresponding to BEGIN and END. So in case the ATM will not sent to the bank the whole amount deposited by the user, these values will not match and the unsafe ATM will be discovered.

Type systems with session types and correspondence assertions can be used to check:

- the source of information,
- whether data are propagated as specified across multiple parties,
- if there are unspecified communications between parties, and
- if *the data being exchanged have been modified* by the code in some unexpected way.

2.5 Reputation

The work [BCCDC11] introduces a notion of reputation into session calculi, with the aim of regulating the participation of principals in sessions depending on the history of their past interactions. To this end, it uses the multirole session calculus of Deniélou and Yoshida [DY11], where each role may be inhabited by a varying number of participants. The reputation associated with principals in a service is built on the basis of their behaviour as participants in past sessions of the service. The service checks the reputation of principals before allowing them to take part in a new session. Symmetrically, principals can declare their own policies, and check them against the reputation of the current participants before deciding whether or not to join the service.

The approach is illustrated by an example describing an online shop, where there are principals who play the role of sellers and principals who play the role of buyers (notice that each principal may play both roles). A (simplified) example of a principal playing the buyer role is the process:

$$s^+! < \mathsf{item} > .s^+? < \mathit{price} > .\text{IF } \mathsf{good}(\mathit{price}) \text{ THEN } s^+!^\bullet < \mathsf{ok} > \text{ ELSE } s^+!^\bullet < \mathsf{ko} >$$

where the sending actions decorated by $^\bullet$ are recorded in the buyer's history. From the point of view of the seller, a buyer who has a long record of purchases will be more interesting than a buyer who has a long record of refusals. Then,

for instance, the seller will be inclined to propose special offers to the first buyer but not to the second, inviting him to join a private sale session.

As for now, this calculus only allows the construction of objective reputations based on histories of session participants. It is therefore suitable for refinements allowing more interesting treatments of reputation, with a closer connection to security and a combination of session types with dynamic typing.

3 Dynamic web data

3.1 Introduction

In an open distributed network, it is extremely important to provide security and protect privacy during transfer and management of data. These issues are reviewed for dynamic web documents handled by XML (Sections 3.2-3.4) and web of data (Linked Data) published and linked using Resource Description Format (RDF) (Section 3.5).

For a given security policy of a distributed system containing semi-structured XML documents, the aim is to provide that the system behaves according to the prescribed security policy. In a calculus with suitable type system, security can be verified by typing, as it is presented in [DCGP06], [DCGPV08], [DCGJP10], [Jak12] and [GJPDC12].

Linked Data provides some sensible guidelines for publishing and consuming data on the Web. Data published on the Web has no inherent truth, yet its quality can often be assessed based on its provenance. Building on [HS11], the paper [DCHS12] provides a calculus of processes which use, consume and publish Linked Data tracing provenance.

3.2 A calculus for modeling dynamic web data

The Xdπ-calculus is introduced in [GM05] as a formal model for reasoning about dynamic web data. A network of peers is modeled as a parallel composition of locations:

$$\mathbf{N} ::= \quad \mathbf{0} \quad | \quad \mathbf{N} \,|||\, \mathbf{N} \quad | \quad l[\![\, T \parallel P \,]\!] \quad | \quad (\nu c)\mathbf{N},$$

where each location consists of a data tree (T) and a process (P). Distinct locations can share communication channels.

The data tree is an unordered edge-labeled rooted tree, with leaves containing empty trees, static (embedded) processes, or pointers:

$$T ::= \emptyset_T \quad | \quad T \mid T \quad | \quad \mathtt{a}[T] \quad | \quad \mathtt{a}[\Box P] \quad | \quad \mathtt{a}[p@l]$$

The pointer $p@l$ points to nodes (i.e. subtrees with the corresponding root nodes) identified by the path p at the location l. Paths are defined by $p ::= \mathtt{a} \quad | \quad p/p \quad | \quad \ldots$ (corresponding to a subset of XPath expressions). The static processes are used for management of semi-structured documents and are spawned by (active) processes.

Processes are:

- the processes for modeling local communication from the π-calculus,
- the process $\mathtt{go}\ l.P$ for movement of processes between locations (from $d\pi$-calculus [HR02]),

- the novel process $\texttt{run}_p$ that locally activates the static processes identified by the path p, and
- the novel process $\texttt{update}_p(\chi, V).P$ that locally updates the local data tree, where χ is a pattern and V is a data term. (We can also assume to have $\texttt{copy}_p(\chi).P = \texttt{update}_p(\chi, \chi).P$ and $\texttt{paste}_p(T).P = \texttt{update}_p(x, x|T).$)

Hence, there are three kinds of interaction: local interaction between processes, local interaction between processes and data trees, and interaction between locations.

3.3 Security of web data

In [DCGP06, DCGPV08], the authors introduce a type system for the Xdπ calculus, in order to control communication of values, access to data and migration of processes between locations. The type system is based on types for locations, data and processes, expressing security levels. *Security levels* are taken from a partially ordered set $(\{i, j, h, \ldots\}, \leq)$. Location names are decorated with security levels. A tree can store in its leaves data of different security levels and it is enclosed in a location of independent security level. The access to data and the mobility of a process depend on the security level of its source location. The source location is the one where the process was in the initial network or where the process was activated from a script.

In a well-typed Xdπ network, the following security properties hold. For security levels i, j and h,

- a channel within a process whose source location has level h can communicate only values whose security levels are less than or equal to h;
- a process whose source location has level h
  - can migrate to a location of level j only if $j \leq h$,
  - can copy from the local tree only data of level j with $j \leq h$, and
  - can modify in the local tree only data of level j with $j < h$;
- a static process of level j which is contained in a leaf of a tree in a location of level i can be activated only if $j \leq i$.

Consider a simple distributed network consisting of an online library, a store, a guest, a member and an owner. Let the library, written in Xdπ notation, be

$$\texttt{library}^1 \; [\![\; T_{\texttt{library}} \parallel P_{\texttt{library}} \;]\!],$$

where $T_{\texttt{library}} = \texttt{author}\ [\texttt{title}[S] \mid \texttt{book}[\texttt{point}[q@\texttt{store}^2] | \texttt{get}\ [\Box P]]]$ and the static process P is the following

$$\texttt{copy}_{\texttt{author/book/point}}(y@x^2).\texttt{go}\ \texttt{x}.\texttt{copy}_y(z^2).\texttt{go}\ \texttt{library}^1.\texttt{paste}_{\texttt{author/book/get}}(z^2)$$

(with patterns decorated with security levels).

Let us assume that the security level of the title S is 1 and the security level of the static process $\Box P$ is 2. If we assign the level 1 to the guest, level 2 to the member and level 3 to the owner, the system behaves according to the following security conditions:

- the guest is allowed to go to the `library` and copy the title, expressed by the process

$$\texttt{go}\ \texttt{library}^1.\texttt{copy}_{\texttt{auhor/title}}(\chi),$$

- the member is allowed to go to the `library` and get the book, by the process

$$\texttt{go}\ \texttt{library}^1.\texttt{run}_{\texttt{author/book/get}},$$

and

- the owner can go to the `library` and change its data by

$$\texttt{go}\ \texttt{library}^1.\texttt{update}_{\texttt{author/book/get}}(\chi, \texttt{abstract}[\emptyset_T]).$$

In this scenario, the access rights can be modified only by changing data.

3.4 Role-based access control of web data

In [DCGJP10], the Xdπ-calculus is equipped with *role-based access control*, with dynamic administration of permissions given to roles, and named ®X$d\pi$-calculus. A lattice of roles is assumed. Each location has a policy and consists of a process with associated roles and a data tree with roles assigned to edge labels, representing permissions (to access edge labels) assigned to roles. Pure (without roles) processes are X$d\pi$ processes extended with commands (`enable` and `disable`) for administration of roles assigned to edge labels. A process is a parallel composition of pure processes with associated sets of roles. A role can be assigned to different edges and different processes, and the behavior of the system is controlled by roles.

A *location policy* is a triple where the first component is the set of minimal roles a process is required to have to access the data at that location. The administration policy is given by the other two components, which prescribe changes of data access rights.

Given a location policy we can check if a data tree and a process comply with it. The type system assures that: if a process can access an edge in a well-typed tree, the edge is connected to the root of the tree by a path whose edges are all accessible to that process; a process can modify a subtree only if it can access all the edges of the subtree; a process can enable a role at an edge or disable a role from a subtree if it can access the path which identifies it. In a well-typed network, all trees and processes in a location comply with the location policy. Moreover, we present some relevant access control properties.

- A channel within a process can communicate only values with at least one (characteristic) role lower than or equal to one role assigned to the process.
- A process
  - can migrate to a location only if it complies with the policy of that location.
  - can read (copy) and change a data in the local tree only if the data is accessible to the process.

– can add (enable) or delete (disable) a role associated to an edge in the local tree only if this is allowed by the location policy.

- A script is activated in a location only if the corresponding process with roles respects the policy of that location.
- A tree built in a location by a change, enable or disable command respects the policy of that location.

For the set of roles $\{\texttt{guest}, \texttt{member}, \texttt{owner}\}$ ordered as $\texttt{guest} \sqsubseteq \texttt{member} \sqsubseteq \texttt{owner}$, consider the online library:

$$\texttt{library}\ [\![\ \texttt{author}^{\{\texttt{guest}\}}\ [T_1|T_2]\ \ \|\ \ R_{\texttt{library}}\ ]\!],$$

where

$$\begin{aligned} T_1 &= \texttt{title}^{\{\texttt{guest}\}}[T], \\ T_2 &= \texttt{book}^{\{\texttt{guest}\}}[\texttt{pointer}^{\{\texttt{member}\}}[T@\texttt{store}]|\texttt{download}^{\{\texttt{member}\}}\ [\Box R]], \end{aligned}$$

and a set of roles is assigned to each edge label (corresponding to XML tags). The edges with the labels $\texttt{author}, \texttt{title}$ and $\texttt{book}$ are accessible to all the roles and all the tree edges are accessible to the roles $\texttt{member}$ and $\texttt{owner}$. The path $\texttt{author/title}$ is accessible to the process with the role $\texttt{guest}$ since both edges are accessible to it, while the path $\texttt{author/book/download}$ is not. In this approach the locations have policies which regulate changes of access rights. For example, if the location policy of the location $\texttt{library}$ is

$$(\{\texttt{guest}\}, \{(\{\texttt{owner}\}, \texttt{guest})\}, \{(\{\texttt{owner}\}, \texttt{member})\})$$

then only processes with a role that is greater or equal to $\texttt{guest}$ can access the library. A process with the role $\texttt{owner}$ can enable the role $\texttt{guest}$ or disable the role $\texttt{member}$. Therefore, having the role $\texttt{owner}$, the process

$$R_{\texttt{library}} = \texttt{enable}_{\texttt{author/book}}(\texttt{guest})^{\neg\{\texttt{owner}\}}$$

gives the permission to access the edges $\texttt{pointer}$ and $\texttt{download}$ in the library to the role $\texttt{guest}$. The subtree T_2 in $\texttt{library}$ becomes:

$$T_2 = \texttt{book}^{\{\texttt{guest}\}}[\texttt{pointer}^{\{\texttt{guest}\}}[T@\texttt{store}]|\texttt{download}^{\{\texttt{guest}\}}\ [\Box R]].$$

In [Jak12], a notion of subtyping is added to the type system and a more flexible type system is obtained.

3.5 A Type System for Provenance Based Access Control

Linked Data recommends the data format to be based on triples of Uniform Resource Identifiers (URIs). The protocols for Linked Data allow triples to be retrieved from locations and written to other locations. Thus a history of 'where and who' provenance can be accumulated. Each time an agent publishes data in a location, the agent and location can be recorded in the provenance history of the triple. Furthermore, the data may be processed locally, by the agent. Recording the operations that were applied to the data provides a notion of 'how' provenance. The paper [DCHS12] introduces a calculus that deals

with *provenance* for Linked Data. For example, the following provenance trace represents that initially there were two pieces of data. One piece of data was published by agent *ACM* in *acm_1*, another piece was published by agent *CiteSeer* in *cs_1*. Agent *RKBExplorer* consumes both pieces of data, applies the function *Clean* to the combination of both pieces of data, and publishes it in location *Toyama*.

$$(RKBExplorer, Toyama) \cdot Clean\# \cdot ((ACM, acm\_1) \vee (CiteSeer, cs\_1))$$

Location are equipped with policies prescribing which agents can read and modify their data.

A type system for the calculus is defined. The type system guarantees that access control policies for data are respected by processes run by agents. The access control policies are based on the provenance of the data. More precisely the typing rules assure that:

1. the provenance traces of the tracked triples agree with the location policies;
2. getting, deleting and inserting operations are always done by agents that are authorised by the location policies.

4 Proof-Carrying Code with Dependent Session Types

4.1 Introduction

Session types consist of high-level specifications of the communication behavior of distributed, concurrent processes along bidirectional channels. Historically, these specifications capture Input/Output behavior, replication (or persistency), branching and selection behaviors, and recursion, enabling static verification of protocol compliance (or session fidelity). However, classic session types are not expressive enough to describe properties of data exchanged in communications, nor to certify such properties in a distributed setting, where the user of a service does not have access to the application source code. Both issues are a fundamental problem in today's world, given the increasing pervasiveness and complexity of distributed services, for which simple descriptions of communication behavior are insufficient characterizations of the rich, high-level contracts these services are intended to follow.

To address the issue of lack of expressiveness in terms of properties that can be characterized by session types, extensions to the session framework have been presented [BCG05] that incorporate *correspondence assertions* into the type structure, enabling reasoning at the level of types through causal dependencies of special events in the executing concurrent processes (e.g. enforcing that a certain set of actions must be preceded by another).

Recently, logical foundations for session types have been established via Curry-Howard correspondences with linear logic [CP10]. Besides clarifying and unifying concepts in session types, such logical underpinnings provide natural means for generalization and extensions. One such extension to *dependent session types* allows us to express and enforce complex properties of data transmitted during sessions [TCP11]. This is achieved by interpreting the first order

quantifiers of intuitionistic linear logic as input and output constructs, in which it is possible to refer to the actual value that is communicated in the types themselves. By combining this with a data language that is itself dependently typed (e.g. in the style of LF [HHP93]), we are able to specify arbitrary properties of the communicated data in such a way that the *proof objects* that witness the desired properties are themselves exchanged during communication. Moreover, the solid logical foundations of the approach allows for further (logically grounded) extensions to the data language to capture features of interest in an almost immediate way, such as digital proof certificates and proof object erasure through modal affirmation and proof irrelevance [PCT11].

4.2 Linear Logic and Dependent Session Types

Linear logic is a logic of resources and evolving state, where propositions can be seen as resources that interact with each other and evolve (i.e. change state) over time. These are the fundamental characteristics that allow for the development of the Curry-Howard correspondence between linear logic and session types.

The work of [CP10] interprets the propositional connectives of linear logic as the session types assigned to π-calculus channels in such a way that linear logic proofs can be interpreted as typing derivations for π-calculus processes. Moreover, the computational procedure of proof simplification or *proof reduction* is directly mapped to inter-process communication, thus obtaining a true correspondence between the dynamics of proofs and the dynamics of communicating processes. The connectives of linear logic are linear implication $A \multimap B$, which is interpreted as the input session type (i.e. input along c a fresh session channel d that will behave as A and proceed along c with the continuation type B); its dual, multiplicative conjunction $A \otimes B$, which is naturally interpreted as session output (output a session channel of type A and continue as B); the multiplicative unit, $\mathbf{1}$, denoting the inactive or terminated session; additive conjunction $A \,\&\, B$ denoting an offer of a choice, meaning that a session of type $A \,\&\, B$ will be able to offer along the session channel either A or B, the choice of which is left to the session client; dually, additive disjunction $A \oplus B$ denotes alternative behavior, and so a session of type $A \oplus B$ will unilaterally choose to behave as either A or B. Finally, the linear logic exponential $!A$ is mapped to replication, in which a session of type $!A$ will offer a potentially unbounded number of instances of the behavior A. Moreover, a fundamental aspect of proof theory is proof composition, also known as a *cut*. In the interpretation, cuts are mapped to *process composition*, where two processes using disjoint sets of resources interact along a fresh session channel, where one offers a session and the other uses it to produce some other session.

Recently, [TCP11] extended this framework of propositional linear logic as session types to incorporate *dependent* session types by moving to a first-order setting, introducing the two quantifiers $\forall x{:}\tau.A$ and $\exists x{:}\tau.A$, where x may occur free in A. The quantification variable is itself typed, with a domain of quantification τ. The language of terms inhabiting τ is a typed λ-calculus, which is left as general as possible, with the usual soundness requirements of progress, substitution and type preservation. The interpretation of these session types is (typed) term output for the existential $\exists x{:}\tau.A$ and term input for the universal $\forall x{:}\tau.A$. Thus, a session of type $\exists x{:}\tau.A$ outputs a term M of type τ and proceeds as type $A\{M/x\}$, whilst a session of type $\forall x{:}\tau.A$ behaves in a dual manner.

By making the quantification domain dependently typed, the authors obtain a session type system where processes exchange data but also proof objects that can denote properties of said data. For instance, the type:

$$\mathsf{UpInterfaceP}(x) \triangleq x : \forall n{:}\mathsf{int}.\forall p{:}n > 0.\exists y{:}\mathsf{int}.\exists q{:}y > 0.\mathbf{1}$$

denotes a session that will input an integer n, a *proof* that n is greater than 0 and will then output back an integer y, itself greater than 0, and a proof of this fact. Well-typedness ensures that these properties hold at runtime due to the existence of these proof objects, making this dependently-typed session framework a *de facto* model of proof-carrying code.

4.3 Proof Irrelevance

In a distributed setting, the proof-carrying framework above requires not only that proof objects exist during type-checking but also enforces that they are transmitted at runtime. However, it is often the case that we want the specified properties to hold but we do not want to exchange the proof objects, either because the properties are easily decidable and the proof objects can be *synthesized* by a decision procedure (for instance, in the example above it is straightforward to check that the communicated numbers are indeed strictly positive) or because the communicating parties have established trust by some external means.

To model the possibility of omitting proofs at runtime, the work of [TCP11, PCT11] extends the framework by internalizing into the proof object language the concept proof theoretical concept of *proof irrelevance* [Pfe01], through a modality which we write $[\tau]$, which types terms of type τ that can be *safely* erased at runtime. This notion of erasure safety essentially means that such terms can never be used to compute values that are not themselves erasable. For instance, the type above can be rewritten as:

$$\mathsf{UpInterfacel}(x) \triangleq x : \forall n{:}\mathsf{int}.\forall p{:}[n > 0].\exists y{:}\mathsf{int}.\exists q{:}[y > 0].\mathbf{1}$$

remarking the fact that the proof objects p and q must be present for type-checking purposes, but they are not used in a computationally significant fashion at runtime and therefore can be safely omitted. This process of erasing proofs at runtime is done in two steps: first we replace all instances of proof irrelevant types and terms with the unit type and element, respectively. Since this does not remove the communication step where the proof objects were previously exchanged, we consistently exploit the type isomorphisms,

$$\forall x{:}\mathsf{unit}.A \cong A$$
$$\exists x{:}\mathsf{unit}.A \cong A$$

to erase the communication overhead in a safe and logically sound way. An alternative technique familiar from type theories is to replace sequences of data communications by a single communication of pairs. When proof objects are involved, these become Σ-types which are inhabited by pairs. For example, we can rewrite the example above as:

$$\mathsf{UpInterfacel}_2(x) \triangleq x : \forall p{:}(\Sigma n{:}\mathsf{int}.[n > 0]).\exists q{:}(\Sigma y{:}\mathsf{int}.[y > 0]).\mathbf{1}$$

This solution is popular in type theory, where $\Sigma x{:}\tau.[\sigma]$ is a formulation of a *subset type*, $\{x{:}\tau \mid \sigma\}$. Conversely, bracket types $[\sigma]$ can be written as $\{x{:}\mathsf{unit} \mid$

$\sigma\}$, except the proof object is always erased. Under some restrictions on σ (i.e. decidability of the underlying theory), subset types can be seen as predicate-based type refinements.

4.4 Affirmation and Digital Certificates

The examples above showcase what can be seen as two extremes in a spectrum of *trust*. In the original example, no trust between the parties is assumed and therefore all proof objects must be made explicit in communication at runtime. On the other hand, proof irrelevance models a scenario of *full trust*, where no proof objects are expected at runtime. In practice, there are trade offs between trust and fully explicit proofs. For instance, when downloading a large application we may be willing to trust its safety if it is digitally signed by a reputable third party, but if we are downloading and running a piece of Javascript code embedded in a web page, we may insist on an explicit proof that it adheres to our security policy. To make these tradeoffs explicit in session types, [PCT11] also incorporates in the framework a notion of *affirmation* (from modal logic) of propositions and proofs by principals. Such affirmations can be realized through explicit digital signatures on proofs by principals, based on some underlying public key infrastructure.

The key component to model these certificates is the addition of a type $\Diamond_K \tau$ to the framework, which types objects that asserting the property τ, signed by principal K using its private key. An affirmation object is built by taking the original proof object that asserts τ and signing it accordingly. Superficially, this may seem redundant insofar as the certificate contains the proof object itself. However, checking a digitally signed certificate may be much faster than checking the validity of a proof, so we may speed up the system if we simply trust K's signature. Moreover, when combining certificates with proof irrelevance, we may construct certificates where parts of the original proof object have been erased, and so we have in general no way of reconstructing the original proofs. In these cases we must necessarily trust the signing principal K to accept τ as true.

Combining affirmation and proof irrelevance it becomes possible to model the following,

$$\mathsf{fpt}{:}\forall f{:}\mathsf{nat} \to \mathsf{nat}.\forall p{:}\Diamond_{\mathsf{verif}}[\Pi x{:}\mathsf{nat}.f(x) \leq x].\exists y{:}\mathsf{nat}.\exists q{:}[y = f(y)].\mathbf{1}$$

which expresses the type of a service that inputs a function f, accepts a verifier's word that it is decreasing (denoted by the object p, which is an certificate of that fact) and returns a fixed point of f to its client. In realistic scenarios such as proof-carrying file systems [GP10], this approach of using affirmation and proof irrelevance results in substantial less overhead in communication when compared to proof-carrying code in the sense of Necula and Lee [Nec97], where the proof objects become too big to be transmitted and checked every time a file is accessed.

4.5 Dynamic Spatial Logics

Reasoning about security often requires describing the structure of systems so as to express the desired properties. For example, one may say that a *secret*

is a piece of information known by a *part* of the system and unforgeable by other parts (from [Cai08a]). Specification logics that allow us to talk about the structure of systems may then be particularly suited to express security properties in a natural way. Furthermore, combining the ability to inspect the structure of systems together with the ability to talk about system behavior, one may then reason about security properties in the context of dynamic concurrent systems.

The relation between dynamic spatial logics (we refer the reader to [Cai08a] for a survey) and behavioural types for security is then twofold. On the one hand, the dynamic properties that characterize system behavior are necessarily related to the behavioural characterizations carried by the types. On the other hand, the structural or spatial characterizations may help describe security properties in a natural way, namely by allowing to talk about separate parts of the system.

In the last years a number of type-based verification approaches based on dynamic spatial logics have been proposed. Focusing on concurrency and resource control, the spatial behavioural types presented in [Cai08b] allow for the description of resource dependencies and resource ownership in a distributed object model. Focusing on communication safety properties, [AB10] combines ideas from dynamic spatial logics and the generic type system presented in [IK04] to analyse properties such as (communication) race freedom, responsiveness and deadlock freedom. The approach presented in [AB10] relies on model-checking performed at the type level, where types provide a spatial-behavioural abstraction of systems, shown decidable for an interesting class of properties in a subsequent work [AB12]. Also related to dynamic spatial logics is the type discipline of behavioural separation introduced in [CS13] for controlling interference in concurrent imperative programming, where types mix both behavioural / temporal and structural / spatial characterizations.

Although security properties are not centrally addressed by the mentioned type systems, they seem to provide an interesting basis to build on so as to reason about security properties in dynamic concurrent systems, given the ability to characterize behavior and spatial distribution which seems like a natural setting to address security properties that talk about what are the allowed behaviors in the several parts of the system.

5 Sessions as security protocol abstractions

5.1 Introduction

Session type systems are able to provide some safety and liveness guarantees for a whole distributed system, as long as all participants are well-typed and the network is trusted. In many realistic settings, however, these assumptions do not hold.

A first approach towards a more realistic scenario is to consider an untrusted network. The solution, currently used in every-day life, is to perform session communications over secure channels, such as the provided by the Transport Layer Security (TLS) protocol. This ensures that well-typed participants will interact safely (precisely, as safely as the TLS protocol allows) within an untrusted enviroment.

A second, more general approach is when some of the (multiparty) session participants are not trusted to be well-typed, i.e., they are not trusted to respect the session specification. This covers those cases in which, for instance, participants rely on implementations provided by non-reliable third-parties, or when they may be controlled by an adversary. In some cases, not respecting the communication pattern (e.g., skipping mandatory messages, not respecting branching) is indeed a security issue. The questions are then the following: what properties can still be ensured for compliant participants? Which cryptography should be used to protect the session? How to make sure that all compliant participants share an identical view of a session execution?

5.2 A secure protocol compiler

Based on a restricted session typed language, Corin et al. [CDF+07, CDF+08] offered a first answer to these questions. Their language, expressed as a local type language with a global graph-like representation, does not support any asynchrony or parallelism—this typed language is sometimes referred to as *sequential multiparty session types*.

The principle of their solution is to use the session specification to generate a cryptographic protocol (and its implementation) that will protect the honest participants against any coalition of compromised peers. The idea is that, in order to ensure that an incoming message is valid with respect to the session specification, that message should carry enough trustworthy information to be able to prove that the protocol history was compliant up to that point. Technically, this is achieved by means of asymmetric cryptography, using signatures of past messages to convince the receiver that the specification was followed by all participants. The minimal (necessary and sufficient) set of signatures to be transmitted and checked are defined in [CDF+07] through the notion of the *visibility*. The protocol also relies on other cryptographic primitives, such as nonces and a cache system to prevent replay attacks between session instances or within a given session.

The formal security notion proved by Corin et al. [CDF+07, CDF+08] is called *session integrity*. It says that the messages received and accepted by all compliant participants are always consistent with correct projected traces of the session specification.

Corin and Deniélou present in [CD07] their prototype implementation as an extension of OCaml. They developed a compiler which takes as input a session description and produces an OCaml module with a function for each participant. Any user code calling one of these functions is guaranteed through the standard ML type system to statically follow the appropriate local session type. This is achieved through a monadic programming style. The module's cryptographic implementation then guarantees that, even in the case of compromised peers, all the messages seen by uncompromised participants are consistent with the session specification.

5.3 Securing a more expressive session language

Planul et al. [PCF09] extend [CDF+08] by considering a more expressive language featuring concurrency and synchronization within session runs. Their

specification language, however, drifts from a session typed language to a CCS-like protocol language whose implementation is represented as a set of traces.

In [BCD+09], Bhargavan et al. take a different approach and rather improve the work in [CD07] with a simpler and more efficient cryptography (using a combination of asymmetric and symmetric cryptography), and extend the session description language with value annotations. This extension allows one to model commitments and to protect independently the integrity of each payload. As in [CD07], a compiler implementation is realised, which relies on OCaml typing for local protocol conformance, and on a generated optimised cryptographic protocol implementation for session integrity.

6 Gradual Security Types and Sessions

While highly expressive fully static type systems can be constructed and proved sound, not many of them have an impact on computing practice. One reason for this deplorable fact is that most software is not written from scratch, but rather by modifying existing components or building on top of them. Clearly, program modifications must be written in the "legacy language" of the existing code. Extensions may be connected to the legacy components by way of foreign function interfaces or by wrapping the legacy code in web services and connecting to it via communication channels. In these cases, the new code may be subject to an expressive type discipline, but the existing code is used as is, because expressive type systems often place structural constraints on the code and it would be too expensive to rewrite existing production code.

This situation aggravates in the security setting because security policies are often stated after the fact, when significant parts of a system have already been implemented, and they are likely to change in reaction to newly discovered threats and exploits or to adhere to new legislative restrictions.

6.1 Gradual Typing and Security

One approach to address these problems is to resort to gradual typing [ST06, ST07], which has received a lot of attention. Gradual type systems are closely related to dynamic type systems and to type systems with type `Dynamic` [ACPP91, Hen94]. In a language with a dynamic type system, a value of arbitrary type T may be injected in a special type `Dynamic`. This injection constructs a pair of a run-time representation of T and the actual value of type T. There are coercions from type `Dynamic` to any other type T, but these coercions may fail if the underlying value does not have type T. Such a system may be embedded in a conventionally typed language [ACPP91] or it may be used to specify and optimize a dynamically typed language [Hen94].

In the extreme case, the language is dynamically typed (like Scheme or JavaScript) and each arithmetic operation, say, has to coerce each argument to a number before it can execute. Among other approaches, coercion calculi have been used to address the inefficiency arising due to type tests where the outcome is known in advance [Hen94]. Gradual typing is another approch to address these efficiency issues. It proceeds by finding a static typing for large parts of a program and by inserting coercions in places where a static type

checker does not succeed. Gradual typing guarantees traditional type safety up to the violations detected by the inserted coercions.

Disney and Flanagan [DF11] have shown that gradual typing may also be applied in a security context, albeit in the setting of the pure lambda calculus. Their approach has later been extended to an ML core language. This extension employs a very liberal treatment of references that are shared between statically and dynamically typed fragments [FT13].

Security type systems that control information flow and that track data integrity usually assume an underlying program that is well-typed according to some standard type system [VIS96]. Security labels added as decorations of the standard types indicate the influence of various peers on the typed value. These labels are mostly drawn from a lattice of confidentiality or integrity levels with at least two distinct points $\bot$ and $\top$ denoting low and high confidentiality, respectively, as already discussed.

Unlike the original work on gradual typing, gradual security typing also assumes an underlying typed program. The gradual aspect of the system is restricted to the security labels, which model secure information flow in the two cited papers. Thus, the property that is guaranteed by the gradual security type system is termination insensitive noninterference [GM82]. The statically security-typed parts have this property by means of the type system, whereas the dynamically security-typed parts have the property by means of a suitable dynamic enforcement mechanism, e.g., a run-time monitor that checks a run-time representation of the security labels.

In the best case, a coercion from static to dynamic adds run-time labels whereas a coercion from dynamic to static removes them. While such a design is possible in the presence of references, it restricts the use of references that are aliased between statically and dynamically typed parts of a program. For that reason, the language proposed by Fennell and Thiemann [FT13] requires some dynamic checks even in the statically typed parts of a program.

Up to this point, the developments support legacy code (which is assumed to be typed, but not with a security type system) embedded in new code developed with the help of a suitable security type system. The gradual approach would place security coercions at the borders of the legacy code, potentially add run-time labels to all values, and monitor its execution. In contrast, the new code would run without labels at full speed because its security properties are guaranteed by the static type system. But the embedding would have the form of a function call, perhaps through a foreign function interface.

Addressing the connection of legacy code with new code via communication channels is the place where session types or other behavioral types enter the scene.

6.2 Gradual Security Typing and Sessions

Wolff and others [WGTA11] have shown that typestate can be subject to gradual typing in the setting of the concurrent object-oriented language Plaid [SNS+11]. The coercion into the dynamic type reifies the current typestate in a run-time value and runs an automaton on it that is synchronized with the transitions of the static typestate computation. They suggest to use the dynamic type during program development as using their full static typestate system is burdensome because it requires many program annotations to manage aliasing.

However, the work on gradual types for Plaid cannot readily be transferred to session types. The main obstacle lies in dealing with the linearity of the channel types, where Plaid resorts to (sophisticated) alias management. However, it has been shown that linearity and gradual typing are largely orthogonal and that many results from standard gradual typing, in particular the blame theorem, carry over to a setting with linear types [FT12]. When restricting to affine typing, then the gradual aspects of the type systems may even be allowed to hide the affine property [TP10].

The significance of a blame theorem is twofold. First, it strengthens the progress property, which is proved as part of a standard type soundness proof, by improving the modeling of stuck (or violation inducing) terms. Second, it clearly locates the demarcation between statically proved and dynamically checked code at a particular kind of coercions, essentially those that coerce from static to dynamic.

Thus, we are now in a position to actually build a system with gradual session types that also supports verifying security properties. It is expected that any of the existing static session systems with security awareness (e.g., [CCDCR10, CCDC11, CCDC13]) can form the basis of a gradual system, similar as in the setting without sessions.

References

[AB10] Lucia Acciai and Michele Boreale. Spatial and behavioral types in the π-calculus. *Inf. Comput.*, 208(10):1118–1153, 2010.

[AB12] Lucia Acciai and Michele Boreale. Deciding safety properties in infinite-state π-calculus via behavioural types. *Inf. Comput.*, 212:92–117, 2012.

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[BBC+06] Michele Boreale, Roberto Bruni, Luís Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, António Ravara, Davide Sangiorgi, Vasco Thudichum Vasconcelos, and Gianluigi Zavattaro. Scc: A service centered calculus. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *Web Services and Formal Methods, Third International Workshop, WS-FM 2006 Vienna, Austria, September 8-9, 2006, Proceedings*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 2006.

[BCCDC11] Viviana Bono, Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. A reputation system for multirole sessions. In Roberto Bruni and Vladimiro Sassone, editors, *Trustworthy Global Computing - 6th International Symposium, TGC 2011, Aachen, Germany, June 9-10, 2011. Revised Selected Papers*, volume 7173 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2011.

[BCD+08] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In *Proc. CONCUR'08*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.

[BCD+09] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, pages 124–140. IEEE Computer Society, 2009.

[BCG05] Eduardo Bonelli, Adriana B. Compagnoni, and Elsa L. Gunter. Correspondence assertions for process synchronization in concurrent communications. *J. Funct. Program.*, 15(2):219–247, 2005.

[BM08] Roberto Bruni and Leonardo Gaetano Mezzina. Types and deadlock freedom in a calculus of services, sessions and pipelines. In José Meseguer and Grigore Rosu, editors, *AMAST*, volume 5140 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2008.

[Cai08a] Luís Caires. Dynamic spatial logics: A tutorial survey. *Bulletin of the EATCS*, 94:77–112, 2008.

[Cai08b] Luís Caires. Spatial-behavioral types for concurrency and resource control in distributed systems. *Theor. Comput. Sci.*, 402(2-3):120–141, 2008.

[CCDC11] Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. Information flow safety in multiparty sessions. In Bas Luttik and Frank Valencia, editors, *Proceedings 18th International Workshop on Expressiveness in Concurrency*, volume 64 of *EPTCS*, pages 16–30, 2011.

[CCDC13] Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. Typing access control and secure information flow in sessions. *Inf. Comput.*, 2013. To appear.

[CCDCR10] Sara Capecchi, Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Tamara Rezk. Session types for access and information flow control. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2010.

[CD07] Ricardo Corin and Pierre-Malo Deniélou. A protocol compiler for secure sessions in ml. In Gilles Barthe and Cédric Fournet, editors, *TGC*, volume 4912 of *Lecture Notes in Computer Science*, pages 276–293. Springer, 2007.

[CDF+07] Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, Karthikeyan Bhargavan, and James J. Leifer. Secure implementations for typed session abstractions. In *20th IEEE Computer Security Foundations Symposium, CSF 2007, 6-8 July 2007, Venice, Italy*, pages 170–186. IEEE Computer Society, 2007.

[CDF+08] Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, Karthikeyan Bhargavan, and James J. Leifer. A secure compiler for session abstractions. *Journal of Computer Security*, 16(5):573–636, 2008.

[CP10] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *21st International Conference on Concurrency Theory*, CONCUR'10, pages 222–236. Springer LNCS 6269, 2010.

[CS13] Luís Caires and João Costa Seco. The type discipline of behavioral separation. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 275–286. ACM, 2013.

[DCGJP10] Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Svetlana Jaksic, and Jovanka Pantovic. Types for role-based access control of dynamic web data. In Julio Mariño, editor, *Functional and Constraint Logic Programming - 19th International Workshop, WFLP 2010, Madrid, Spain, January 17, 2010. Revised Selected Papers*, volume 6559 of *Lecture Notes in Computer Science*, pages 1–29. Springer, 2010.

[DCGP06] Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, and Jovanka Pantovic. Security types for dynamic web data. In Ugo Montanari, Donald Sannella, and Roberto Bruni, editors, *TGC*, volume 4661 of *Lecture Notes in Computer Science*, pages 263–280. Springer, 2006.

[DCGPV08] Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Jovanka Pantovic, and Daniele Varacca. Security types for dynamic web data. *Theor. Comput. Sci.*, 402(2-3):156–171, 2008.

[DCHS12] Mariangiola Dezani-Ciancaglini, Ross Horne, and Vladimiro Sassone. Tracing where and who provenance in linked data: A calculus. *Theor. Comput. Sci.*, 464:113–129, 2012.

[DF11] Tim Disney and Cormac Flanagan. Gradual information flow typing. In *STOP 2011*, 2011.

[DY11] Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic Multirole Session Types. In Mooly Sagiv, editor, *Proc. POPL'11*, pages 435–446. ACM, 2011.

[FT12] Luminous Fennell and Peter Thiemann. The blame theorem for a linear lambda calculus with type dynamic. In *Trends in Functional Programming*, volume 7829 of *Lecture Notes in Computer Science*. Springer, 2012.

[FT13] Luminous Fennell and Peter Thiemann. Gradual security typing with references. In Véronique Cortier and Anupam Datta, editors, *CSF*, pages 224–239, New Orleans, LA, USA, 2013. IEEE.

[GJ03] Andrew D. Gordon and Alan Jeffrey. Typing correspondence assertions for communication protocols. *Theor. Comput. Sci.*, 300(1-3):379–409, 2003.

[GJPDC12] Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, and Mariangiola Dezani-Ciancaglini. Types and roles for web security. *Transactions on Advanced Research*, 8(2):16–21, 2012.

[GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[GM05] Philippa Gardner and Sergio Maffeis. Modelling dynamic web data. *Theor. Comput. Sci.*, 342(1):104–131, 2005.

[GP10] Deepak Garg and Frank Pfenning. A proof-carrying file system. In D.Evans and G.Vigna, editors, *Proceedings of the 31st Symposium on Security and Privacy (Oakland 2010)*, Berkeley, California, May 2010. IEEE. Extended version available as Technical Report CMU-CS-09-123, June 2009.

[Hen94] Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22:197–230, 1994.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. ACM*, 40:143–184, January 1993.

[Hon93] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.

[HR02] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Inf. Comput.*, 173(1):82–120, 2002.

[HS11] Ross Horne and Vladimiro Sassone. A typed model for Linked Data. Technical report, University of Southampton, 2011. `http://eprints.ecs.soton.ac.uk/21996/`.

[HVK98] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *Proc. POPL'08*, pages 273–284. ACM Press, 2008.

[IK04] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.

[Jak12] Svetlana Jaksic. Input/output types for dynamic web data. In *Theoretical Computer Science, 13th Italian Conference, ICTCS 2012, Varese, Italy, September 19-21, 2012, Proceedings*, 2012.

[Kob02] Naoki Kobayashi. A Type System for Lock-Free Processes. *Information and Computation*, 177:122–159, 2002.

[Kol08] Marija Kolundzija. Security types for sessions and pipelines. In Roberto Bruni and Karsten Wolf, editors, *Web Services and Formal Methods, 5th International Workshop, WS-FM 2008, Milan, Italy, September 4-5, 2008, Revised Selected Papers*, volume 5387 of *Lecture Notes in Computer Science*, pages 175–190. Springer, 2008.

[LPT07] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. Regulating data exchange in service oriented applications. In Farhad Arbab and Marjan Sirjani, editors, *International Symposium on Fundamentals of Software Engineering, International Symposium, FSEN 2007, Tehran, Iran, April 17-19, 2007, Proceedings*, volume 4767 of *Lecture Notes in Computer Science*, pages 223–239. Springer, 2007.

[Nec97] George C. Necula. Proof-carrying code. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 106–119. ACM Press, 1997.

[Pad13] Luca Padovani. From Lock Freedom to Progress Using Session Types. In *PLACES'13*, EPTCS, 2013. to appear.

[PCF09] Jérémy Planul, Ricardo Corin, and Cédric Fournet. Secure enforcement for global process specifications. In Mario Bravetti and Gianluigi Zavattaro, editors, *CONCUR 2009 - Concurrency Theory, 20th International Conference, CONCUR 2009, Bologna, Italy, September 1-4, 2009. Proceedings*, volume 5710 of *Lecture Notes in Computer Science*, pages 511–526. Springer, 2009.

[PCT11] Frank Pfenning, Luís Caires, and Bernardo Toninho. Proof-carrying code in a session-typed process calculus. In *CPP*, pages 21–36, 2011.

[Pfe01] F. Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *16th Symposium on Logic in Computer Science*, LICS'01, pages 221–230. IEEE Computer Society, 2001.

[PSS05] François Pottier, Christian Skalka, and Scott F. Smith. A systematic approach to static access control. *ACM Trans. Program. Lang. Syst.*, 27(2):344–382, 2005.

[PT12] Rosario Pugliese and Francesco Tiezzi. A calculus for orchestration of web services. *J. Applied Logic*, 10(1):2–31, 2012.

[SNS+11] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in Plaid. In Cristina Videira Lopes and Kathleen Fisher, editors, *OOPSLA*, pages 713–732. ACM, 2011.

[ST06] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.

[ST07] Jeremy Siek and Walid Taha. Gradual typing for objects. In Erik Ernst, editor, *21st European Conference on Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Germany, July 2007. Springer-Verlag.

[TCP11] Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent session types via intuitionistic linear type theory. In *Proceedings of the 13th International Symposium on Principles and Practice of Declarative Programming (PPDP'11)*, pages 161–172. ACM, July 2011.

[TP10] Jesse A. Tov and Riccardo Pucella. Stateful contracts for affine types. In Andrew D. Gordon, editor, *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 550–569. Springer, 2010.

[VIS96] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.

[WGTA11] Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In Mira Mezini, editor, *ECOOP*, volume 6813 of *Lecture Notes in Computer Science*, pages 459–483, Lancaster, UK, 2011. Springer.