Combining behavioural types with security analysis

Massimo Bartoletti[a], Ilaria Castellani[b], Pierre-Malo Deniélou[c], Mariangiola Dezani-Ciancaglini[d], Silvia Ghilezan[e], Jovanka Pantovic[e], Jorge A. Pérez[f], Peter Thiemann[g], Bernardo Toninho[h], Hugo Torres Vieira[i]

[a] *Dipartimento di Matematica e Informatica, University of Cagliari, Italy*
[b] *INRIA Sophia Antipolis, France*
[c] *Royal Holloway, University of London, UK*[1]
[d] *Dipartimento di Informatica, Università di Torino, Italy*
[e] *Faculty of Technical Sciences, University of Novi Sad, Serbia*
[f] *University of Groningen, The Netherlands*
[g] *University of Freiburg, Germany*
[h] *Department of Computing, Imperial College London, UK*
[i] *IMT Institute for Advanced Studies Lucca, Italy*

Abstract

Today's software systems are highly distributed and interconnected, and they increasingly rely on communication to achieve their goals; due to their societal importance, security and trustworthiness are crucial aspects for the correctness of these systems. Behavioural types, which extend data types by describing also the structured behaviour of programs, are a widely studied approach to the enforcement of correctness properties in communicating systems. This paper offers a unified overview of proposals based on behavioural types which are aimed at the analysis of security properties.

1. Introduction

Computing systems are omnipresent nowadays; besides their classical application domains, they have entered into multiple dimensions of our lives, from business to leisure, from finance to e-government and health, from global logistics to home appliances, to mention but a few. Most of these systems are distributed over the network, and thus rely heavily on *communication* to carry out their tasks; for example, in the financial world where decisions are taken in the global market, or in the context of emerging home appliances that autonomously shop for groceries on our behalf. Given their importance and societal impact, it is crucial that these communicating systems behave in a reliable way. This is not an easy task, since they have to run over *open networks*, where they can be targeted by malicious parties trying to threaten their functionality, or to seize or compromise sensitive data. It is therefore fundamental to develop rigorous (and scalable) techniques to ensure the reliability and security of these systems.

Distributed systems are very challenging to analyse, for a variety of reasons: these range from their intrinsic heterogeneous nature, to the possible presence of untrusted

[1] Now at Google Inc.

components, to the complexity of the interactions and of their induced behaviours. In the realm of programming languages, *type systems* represent a well-established technique to ensure program properties. Types allow programmers to single out programs that are correct (i.e., error-free, for a certain class of errors) at compile-time, just by inspecting their source code. Examples of errors that may be excluded by type systems are the inability of an object to handle a method call (message not understood), *races* (competition among concurrent programs for some shared resource), which may lead to inconsistent states or unexpected behaviours, and *communication errors*, caused by non-matching expectations of two communicating partners.

In today's open and highly distributed computing environment, security flaws of various kinds may arise, and it is crucial to exclude them before programs are deployed. Static and dynamic techniques for ensuring access control and secure information flow were originally conceived for operating systems. In the last two decades, spurred by the pioneering work of [33] and [93] on static analysis for secure information flow, type systems targeting security properties have been gradually introduced both into specification languages such as process calculi [19, 31, 43, 55, 56, 59, 60, 66, 77, 80] and into full fledged programming languages [6, 27, 53, 81, 84].

Classical data types are an abstract specification of *what* programs compute (i.e., the outcome of computations). By contrast, *behavioural types* (*BTs*) specify also *how* programs compute (i.e., the structure of computations) thus giving a more intensional description of their behaviour (the reader is referred to [62] for an extensive discussion on this point). BTs are particularly suited in dealing with concurrent and communicating processes, whose semantics is based on their *interactive behaviour*. In this case, BTs may be viewed as abstract protocols, describing the causal and branching structure of communications among a number of parties. First introduced in the setting of process calculi [57, 58], behavioural types for communication-centric systems have been studied for a variety of calculi and languages since the late nineties. For a recent survey, the reader is referred to [2].

The design of behaviourally typed languages requires some care: on the one hand, one would like to keep a simple and intuitive programming flavour (to ensure that typed abstractions may be widely translated into practice); on the other hand, typed languages must be developed bearing in mind the complexity (not to mention decidability) of their associated verification techniques. Hence, BT-based approaches must be fine-tuned to keep a balance between expressiveness and feasibility of analysis.

In this respect, *session types* [57, 58] stand out as a particularly attractive instance of behavioural types. Session types allow interactions to be structured into basic units called *sessions*. Individual interaction patterns are then abstracted as session types, against which process descriptions may be checked. The expressiveness of session types has enabled their application in diverse contexts, targeting different programming models (e.g., functional [92] and object-oriented programming [34]), and addressing also lower-levels of application (namely operating system design [39] and middleware communication protocols [91]), to mention a few.

Although some proposals that promote the use of BTs for the analysis of security properties have been put forward, the study of security and trustworthiness properties for typed communicating systems is still at an early stage.

Security type systems for variants of the π-calculus, which combine security en-

forcement with other correctness concerns, were presented in [55, 56] and [59, 60]. A simple type system ensuring noninterference for the π-calculus was proposed in [77]. Subsequently, more refined security type systems for the π-calculus were studied in [31] and [66]. This last work provides a sophisticated security analysis, together with a type inference algorithm.

The question of unifying the language and process-based approaches to security has also been addressed. The goal was to define translations from the former to the latter that would preserve both the security properties and the security types. A first step in this direction was taken in [59], where a typed parallel imperative language was embedded into a typed π-calculus. This work was further pursued in [60], where more powerful languages, both imperative and functional, were considered. Among the types used in these early works, the closest to behavioural types are the *usage types* of [66], which describe how processes use their communication channels along computations. It should be noted, however, that all these approaches are based on (simple variants of) the π-calculus, and not on so-called *session calculi*, more specialised variants of the π-calculus where communication has an explicit logical structure.

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 (restricted) channel of the session. In general, one expects security properties to be simpler to enforce within a session than in an open network: since session participants must conform to their session types, their behaviour is more disciplined than that of arbitrary, untyped processes. Indeed, in a trusted session-typed setting, one may focus on systems where participants communicate according to prescribed protocols – this property is often referred to as *session fidelity* – and this restricts the range of possible security flaws. Moreover, within this trusted platform, one may address more specific security properties, regarding, e.g., the information that is communicated or the participants who carry out the communications, in order to achieve a finer tracking of security flaws. Finally, since interactions within a session take place on a private channel, session isolation is guaranteed by construction.

However, this scenario is too restrictive in practice. In realistic distributed applications, the components are spread across open networks and cannot be completely controlled. Hence, both the *closed-network assumption* (ensuring the isolation of session channels) and *the typed-world assumption* (ensuring the correct behaviour of all components) have to be lifted. Only part of the application may be assumed to be typed, and hence trusted. The challenge is then to guarantee that the desired security properties continue to hold when the trusted part interacts with the untrusted one.

In this document, we present a survey of existing BT-based approaches aimed at ensuring security properties. Our review does not only reflect several different views of secure and trustworthy communication-based systems, but also highlights how behavioural types provide a simple conceptual framework to formally approach all such different visions, from various angles.

The rest of the paper is organised as follows. Section 2 introduces the basic notions of session types. In Section 3 we discuss behavioural types which guarantee access control, secure information flow, and integrity of data in communication protocols. Section 4 shows how the logical foundation of session types – based on the correspondence with linear logic [21] – can be fruitful also for security. In particular, dependent

session types are applied to proof carrying code and digital certificates.

To some extent, it is fair to say that the approaches described in Sections 3–4 consist in extending foundational settings with security concerns or enhancing existing techniques in order to tackle security issues. In a somewhat distinct direction, we find techniques that focus on guaranteeing that the properties studied in foundational settings can be transported to more realistic models encompassing open networks: Section 5 reviews a compilation of session-like descriptions into implementations of cryptographic protocols that ensures that honest session participants are protected from external interference. Section 5 also describes a theory of contracts that addresses a different notion of honesty among interacting parties, roughly referring to participants that behave as promised (contracted) even when engaged in other interactions with dishonest parties. The protection of participants from malicious contexts is further addressed in Section 5.4, by means of game theoretic approaches.

Section 6 presents recent developments, including two approaches that use dynamic typing, in contrast to the type systems reviewed in previous sections. Gradual typing allows legacy code to be reconciled with varying security policies (Sections 6.1 and 6.2). Section 6.3 describes a process framework which exploits behavioural types to jointly enforce run-time adaptation and the combination of access control and secure information flow. Section 6.4 reports on a combination of techniques that address access control with a behavioural typing system that focuses on role-based protocol specifications, considering that role impersonation must be duly *authorized*. Section 7 summarises the various approaches reviewed in the paper, and gives some tracks for future research.

2. Preliminaries

In this section, we informally present some key notions pertaining to behavioural types, focusing on session types, which are the kind of BTs for which most security-oriented extensions have been proposed. The interested reader is referred to the recent survey [62] for further details on the foundations of behavioural types.

Session types characterise how communication channels are used by programs. Intuitively, much like the type declaration *var*: `int` tells us that *var* will be used to hold integer values, a type declaration chan: `SessionType` tells us that chan will be used to hold an access point to a channel that will be used by the program according to `SessionType` . However, `SessionType` may actually refer to several stages of the usage of chan. A simple instance of a session type is

$$\mathtt{S}_1 = \mathtt{!int.?bool.end}$$

Then, the type declaration chan : $\mathtt{S}_1$ indicates that chan is first used to output an integer and then to input a boolean. More precisely, '`!`' denotes output, '`?`' represents input, '`.`' captures sequentiality, and '`end`' denotes no further usage.

A fundamental feature of session types is that they capture *linear* interactions. Linearity amounts to forbidding competing communications on the same channel, what is sometimes referred to as a *communication race*. Session types exclude communication races but they allow for the specification of alternative behaviours (controlled by one

party), described via so-called *branching* and *selection* types. Branching represents an *external choice*, to be solved by the communicating partner, while selection represents an *internal choice*, typically placed in the branch of a conditional. Hence, alternatives are modelled by a "menu" and a corresponding choice: the former is usually implemented via a set of available inputs, and typed using branching; the latter is usually implemented via an output, and typed using selection. Alternatives are conveniently identified by *labels*. For example, the session type

$$\mathtt{S}_2 = \text{NUMBER}\ \mathtt{?int.!int.end} + \text{CONDITION}\ \mathtt{?bool.end}$$

may be used to describe a process that is waiting for the choice of either one of its offered options, identified by labels NUMBER and CONDITION. This process can be safely composed with, e.g., a process willing to first choose the NUMBER option and then to send and receive an integer.

It is worth noticing that since types talk about the current stage of the protocol, type preservation, i.e., the property that ensures that well-typedness is preserved under system evolution, holds modulo an evolution also of types (types are resources that in some sense get consumed by execution).

Another powerful feature of session types is *(session) delegation*, which allows third parties to gain access to an already established interaction. Realised via channel passing, this mechanism is useful to specify, e.g., a server that at some stage in the protocol delegates to a third party the remaining communications with a client (who will proceed with the protocol, unaware of the exchanges at the server side). Carried types can thus be session types themselves; this way, e.g., the session type

$$\mathtt{S}_3 = \mathtt{?(!int.end).end}$$

specifies the reception of a channel, which will be used to output an integer according to $\mathtt{!int.end}$.

Session types for binary "client-server" interaction, as introduced in [58], capture the two-ended interaction via the type of one of the endpoints — which are sometimes distinguished by so-called *polarities* [46], denoted $+$ and $-$: this way, e.g., chan^+ and chan^- would denote the two endpoints of chan. So, if one endpoint is used according to a session type, it is immediate to recover the characterisation of the other endpoint by replacing inputs by outputs, branching by selection and conversely — a notion usually referred to as *duality*. The dual of a session type $\mathtt{S}$ is denoted $\overline{\mathtt{S}}$. This way, e.g., the duals of session types $\mathtt{S}_1$, $\mathtt{S}_2$ and $\mathtt{S}_3$ above are respectively:

$$\overline{\mathtt{S}_1} = \mathtt{?int.!bool.end}$$
$$\overline{\mathtt{S}_2} = \text{NUMBER}\ \mathtt{!int.?int.end} \oplus \text{CONDITION}\ \mathtt{!bool.end}$$
$$\overline{\mathtt{S}_3} = \mathtt{!(!int.end).end}$$

More recently, following the approach introduced in [61], generalisations of session types that capture multiparty interactions have been put forward. In such a setting of *multiparty sessions*, pairwise linear interaction is still ensured, via the use of dedicated intra-session channels (or session indexes [13]). In this case, the session types of individual parties no longer suffice to capture the entire structured interaction: global

specifications, called *global types*, have been introduced to specify sequencing information for message exchanges among the various communicating parties. Since a global type explicitly identifies the involved parties, it is possible to extract from it the individual contributions of each party via a *projection* function — these contributions are commonly referred to as *local types*.

In the above examples and in the sequel the syntactic categories are distinguished by using different fonts, namely: `sort types, session types, contracts`, CHOICE LABELS , values, participants, *variables, processes, terms*, **`keywords`**.

3. Security Types for Communication-Centred Calculi

An increasingly relevant security issue is that of preserving the *confidentiality* of private data that is hosted on cloud infrastructures and/or manipulated by Web services and 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 of legally accessed data to untrusted users, thus ensuring end-to-end confidentiality. Compared to access control, secure information flow may be viewed as additionally restricting the access to transferred or transformed data, when these have been computed using sensitive data. Type systems for access control and secure information flow are reviewed in Sections 3.1 and 3.2, respectively.

Another important security property is data *integrity*, which is often presented as the dual of confidentiality and may be similarly expressed as a combination of access control and secure information flow. While confidentiality requires that data should not be released to untrusted destinations, integrity requires that data should not be affected by untrusted parties. Type systems for data integrity are discussed in Section 3.3.

3.1. Access Control

An early work featuring behavioural types for access control is [68], which presents a type system for COWS (Calculus for Orchestration of Web Services) [79], a formalism for specifying and combining services, as well as modelling their dynamic behaviour. The COWS language provides a primitive for killing processes, possibly provoking the abortion of ongoing sessions. This type system allows for the specification and enforcement of *policies* for regulating the exchange of data among services. To implement such policies, programmers can annotate data with sets of participants authorised to use and exchange these data. The typed operational semantics uses these annotations to guarantee that computations proceed correctly.

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

1. the buyer asks the seller for some item and receives back a price;
2. the buyer may either accept the price and send a credit card number to the bank, or turn down the offer.

In this scenario, the desired policy is that the credit card number should be accessible only to the buyer and the bank, but not to the seller. Therefore, the type system of [68] validates processes implementing the protocol described above, but not variants of it in which by mistake the buyer would send the credit card number to the seller. This

policy is represented by decorating the data creditCardNumber with participants agent and bank: $\{\mathsf{creditCardNumber}\}_{\mathsf{agent,bank}}$.

Another related service-oriented calculus is SCC (Service Centered Calculus) [18]. The work [67] enriches a variant of SCC, investigated in [20], with security levels for controlling access rights. In the original calculus, communications may either follow fixed protocols or use pipelines. A *pipe constructor* $P < x < Q$, similar to that of the Orc language [64], replaces the variable x in (a freshly spawned copy of) the process Q with a value sent by the process P. In the new calculus, processes are *framed* [78] 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 [67] comes equipped with a behavioural type system that statically ensures these security properties.

In the buyer-seller-bank protocol described above, the protection of the credit card number is ensured by giving it a security level which is incomparable with the level of the seller, but is smaller or equal to the levels of the buyer and the bank. More formally, if the processes representing the seller, the buyer and the bank are framed by the security levels ℓ_1, ℓ_2 and ℓ_3, respectively, then it is enough to assign to the credit card number the type `nat` with some level ℓ such that $\ell \not\leq \ell_1$, $\ell \leq \ell_2$ and $\ell \leq \ell_3$.

The work [22] adopts the same treatment of access control as [67], ascribing security levels to both participants and data, but it gains flexibility thanks to the mechanism of *delegation*. Namely, it allows participants to get around access control restrictions by delegating the handling of sensitive data to other participants with higher credentials. For instance, instead of sending a bank connection to the buyer, thus explicitly involving the bank in the interaction, the seller may delegate to the bank the part of the interaction that deals with the credit card in a way that is transparent to the client. The type system that ensures access control has an explicit type constructor to track delegation; it allows the delegated part of a session type to be marked.

Notably, all the types investigated in [68], [67], and [22] are behavioural types. While those used in [22] are standard session types enhanced with security constraints, the others are more general instances of BTs. A comparison between the three access control approaches described above is not easy, since the underlying calculi offer different interaction patterns, namely a primitive for session killing in [68], a pipeline constructor in [67] and channel delegation in [22]. In our view, the most challenging construct for data protection is delegation, since it allows a transparent change of ownership for a given communication channel.

3.2. Secure Information Flow

As already mentioned above, the work [22] considers a calculus for multiparty sessions with delegation, enriched with security levels for both participants and data, and equipped with an access control mechanism. This work also defines a secure information flow property, formalising the preservation of data confidentiality. Finally, a session type system is proposed, which introduces secure information flow

requirements in the typing rules, in order to simultaneously ensure the noninterference property and the standard behavioural properties prescribed by session types. Such security-enhanced session types are an instance of behavioural types specifying both the sequencing of communication actions and the constraints between their security levels. The study [22] 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 essence, session-typed processes are less prone to security flaws. This point will be discussed in some depth later in this section. We now describe the approach of [22] in more detail.

The *secure information flow* property defined in [22] is based on the observation of messages while they are being exchanged. As usual in the secure information flow literature, 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 $\bot$ (although [22] deals with a general lattice of security levels). A message or I/O communication action whose carried value is of security level $\top$ (respectively, $\bot$) will be called "secret" or "high" (respectively, "public" or "low").

We shall use here a simple syntax where $\mathsf{c}(x^\ell).P$ denotes the input on channel c of a value of level ℓ, to be replaced for x^ℓ in the continuation process P, and $\mathsf{c} < \mathsf{v}^\ell > .P$ denotes the output on channel c of the value v of level ℓ, to be followed by P. For simplicity, we shall only deal with binary sessions here, so c will range over binary session endpoints of the form $\mathsf{s}^+, \mathsf{s}^-$, where s is the name of some established session.

Secure information flow is usually formalised via the notion of *noninterference* [50]. Noninterference essentially means that low outputs should not depend on high inputs. Then, a typical insecure information flow, also called *information leak*, arises when different high inputs cause different low messages to be exchanged, as in the following process, where we assume $x^\top$ to be a boolean variable:

$$\mathbf{if}\ x^\top\ \mathbf{then}\ \mathsf{s}^+!<1^\bot>\ \mathbf{else}\ \mathsf{s}^+!<2^\bot>$$

Another source of information leaking is the possible blocking of a high input action, in the case where the environment does not offer the expected high message (which is something one cannot control, since two environments differing only for the presence or absence of a high message cannot be distinguished by a low observer). For instance, the process:

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

emits the low output "1" only if it first receives the high input from the environment. Assuming again $x^\top$ to be a boolean variable, the session type of channel s^+ in the above process is `?bool.!int.end`. On the other hand, this process cannot be typed in the security-enhanced session type system, since it exhibits a "level drop" from the input to the output.

The subsequent paper [23] 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. This monitored semantics induces a safety property: a process is safe if none of its computations is blocked by the monitored semantics. This property is called *information flow safety*, and it is proved to strictly imply the

noninterference property of [22]. This is expected, since information flow safety requires the successful execution of all individual computations, while noninterference is a property of the set of computations of a process, which may hold even if some of the computations exhibit information leaks.

The approach in [22, 23] 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 difference between typability, safety and security in [22, 23] by means of a simple example that should convey the appropriate intuitions. In this example, 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.

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

$$\mathbf{if}\ \mathsf{true}^{\top}\ \mathbf{then}\ \mathsf{s}^{+}!\langle 1^{\bot}\rangle\ \mathbf{else}\ \mathsf{s}^{+}!\langle 2^{\top}\rangle$$

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:

$$\mathbf{if}\ \mathsf{true}^{\top}\ \mathbf{then}\ \mathsf{s}^{+}!\langle 2^{\top}\rangle\ \mathbf{else}\ \mathsf{s}^{+}!\langle 1^{\bot}\rangle$$

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 in any computation. An example of a typable process is:

$$\mathbf{if}\ \mathsf{true}^{\top}\ \mathbf{then}\ \mathsf{s}^{+}!\langle 1^{\top}\rangle\ \mathbf{else}\ \mathsf{s}^{+}!\langle 2^{\top}\rangle$$

In this process, channel s^{+} has the security-enhanced type $!\mathtt{int}^{\top}.\mathtt{end}$.

Discussion

There appears to be an influence of classical session types [61] upon security types [93]. Indeed, one of the causes of insecure information flow in a concurrency scenario is the possibility of different *termination behaviours* in the branches of a high conditional (i.e., a conditional which tests a high expression). This may give rise to the 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 to contain this phenomenon, by imposing some uniformity in the termination behaviours of conditional branches: for instance, a terminating branch cannot coexist with a diverging branch, as exemplified below. 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.

Note that one may also use typing to ensure (proper) termination. For instance, termination is ensured by the session type system studied in [21], as shown in [73]. The system in [21] is the basis for the dependent session types reviewed in Section 4.

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

$$\mathbf{if}\ x^{\top}\ \mathbf{then}\ \mathsf{s}^{+}!<1^{\top}>\ \mathbf{else}\ \mathbf{rec}\ X.\mathsf{s}^{+}!<2^{\top}>.X$$

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

$$\mathbf{if}\ x^{\top}\ \mathbf{then}\ \mathsf{s}^{+}!<1^{\top}>\ \mathbf{else}\ \mathsf{s}^{+}!<x^{\top}+3>$$

where we assume that $x^{\top}$ is replaced by a boolean value. However, the security-enhanced typing considered in [22, 23] does not prevent *global deadlocks* due to bad matchings of protocols in interleaved sessions, like in the process:

$$\begin{array}{l}\mathbf{if}\ x^{\top}\ \mathbf{then}\ \mathsf{s}^{+}?(y).\mathsf{s}^{+}!<1^{\top}>.\mathsf{r}^{+}?(z).\mathsf{r}^{+}!<2^{\top}>\\ \qquad\mathbf{else}\ \mathsf{s}^{+}?(y).\mathsf{r}^{+}?(z).\mathsf{s}^{+}!<1^{\top}>.\mathsf{r}^{+}!<2^{\top}>\\ |\ \mathsf{s}^{-}!<3^{\top}>.\mathsf{s}^{-}?(t).\mathsf{r}^{-}!<4^{\top}>.\mathsf{r}^{-}?(u)\end{array}$$

Here, if the **then** branch is taken the process will terminate successfully, while if the **else** branch is taken the interaction will deadlock. The security-enhanced session type of the channels s^{+}, r^{+} is $\texttt{?int}^{\top}\texttt{.!int}^{\top}\texttt{.end}$, while the channels s^{-}, r^{-} have the dual session type $\texttt{!int}^{\top}\texttt{.?int}^{\top}\texttt{.end}$. These global deadlocks are forbidden by the more refined behavioural type systems in [13, 65, 72].

3.3. Integrity of Communicated Data

We now turn to the issue of data integrity, as investigated in the work [17]. We start by considering the standard User-ATM-Bank example [58]. 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:

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

can be typed since channels s^{+}, r^{+} have type $\texttt{!string.!int.end}$, while the channels s^{-}, r^{-} have the dual session type $\texttt{?string.?int.end}$.

In order to cope with this kind of misbehaviour, *correspondence assertions* [51] are incorporated in the theory of session types [17]. Two correspondence assertions can be paired by the keywords **begin**, **end** and their values 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:

$$\mathsf{user} = \mathbf{begin}(\mathsf{userId}, \mathsf{depositAmount}).\mathsf{s}^{+}! < \mathsf{userId} > .\mathsf{s}^{+}! < \mathsf{depositAmount} >$$
$$\mathsf{bank} = \mathsf{r}^{-}?(\mathit{userId}).\mathsf{r}^{-}?(\mathit{depositAmount}).\mathbf{end}(\mathit{userId}, \mathit{depositAmount})$$

thus allowing the malicious ATM to be discovered, since the operational semantics requires the same values in paired correspondence assertions. The session types of the channels remain unchanged, given that correspondence assertions play the role of run-time monitoring.

Compared to standard session types, session types enhanced with correspondence assertions may be used to check additional properties, namely:

- 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* in some unexpected way.

More recently, [4] presents a π-calculus with assume and assert operations, typed using a session discipline that incorporates refinement formulae written in a fragment of Multiplicative Linear Logic [49]. This original combination of session and refinement types, together with the well-established benefits of linearity, allows very fine-grained specifications of communication protocols in which refinement formulae are treated as logical resources rather than persistent truths.

Another related paper is [16], where session types for multiparty sessions are enriched with constraints on the content of the exchanged messages, conditions on the choice of sub-conversations to follow, and invariants on recursion.

4. Logical Approaches to Security based on Behavioural Types

As already discussed, session types consist of high-level specifications of the communication behaviour of distributed, concurrent processes along channels. Historically, these specifications capture input/output behaviour, replication (or persistency), branching and selection behaviours, 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 behaviour are insufficient characterisations 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 characterised by session types, extensions to the session framework have been presented (e.g., the work [17], already discussed in section 3.3, and [86]). Recently, logical foundations for session types have been established via Curry-Howard correspondences with linear logic [21]. Besides clarifying and unifying concepts in session

types, such logical underpinnings provide natural means for generalisation and extensions. One such extension to *dependent session types* allows for expressing and enforcing complex properties of data transmitted during sessions [89]. 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 [52]), 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 enable 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 proof irrelevance and affirmation modalities [75].

Sections 4.1, 4.2, and 4.3 overview extensions of session types with (value) dependent types, proof irrelevance, and affirmation, respectively.

4.1. 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 [21] 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 $\mathtt{A} \multimap \mathtt{B}$, which is interpreted as the input session type (i.e., input a session channel of type $\mathtt{A}$ and continue as $\mathtt{B}$); its dual, multiplicative conjunction $\mathtt{A} \otimes \mathtt{B}$, which is naturally interpreted as session output (i.e., output a session channel of type $\mathtt{A}$ and continue as $\mathtt{B}$); the multiplicative unit, $\mathbf{1}$, denoting the inactive or terminated session; additive conjunction $\mathtt{A} \,\&\, \mathtt{B}$ denoting an offer of a choice, meaning that a session of type $\mathtt{A} \,\&\, \mathtt{B}$ will be able to offer along the session channel either $\mathtt{A}$ or $\mathtt{B}$, the choice of which is left to the session client; dually, additive disjunction $\mathtt{A} \oplus \mathtt{B}$ denotes alternative behaviour, and so a session of type $\mathtt{A} \oplus \mathtt{B}$ will unilaterally choose to behave as either $\mathtt{A}$ or $\mathtt{B}$. Finally, the linear logic exponential $!\mathtt{A}$ is mapped to replication, in which a session of type $!\mathtt{A}$ will offer a potentially unbounded number of instances of the behaviour $\mathtt{A}$. Moreover, a fundamental aspect of proof theory is proof composition, also known as a *cut*. In the interpretation, cuts are mapped to *process composition*; 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 behaviour.

Recently, [89] 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 $\forall x{:}\tau.\mathtt{A}$ and $\exists x{:}\tau.\mathtt{A}$, where x may occur free in $\mathtt{A}$. 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 $\exists x{:}\tau.\mathtt{A}$ and term input for the universal $\forall x{:}\tau.\mathtt{A}$. Thus, a session of type $\exists x{:}\tau.\mathtt{A}$ outputs a term M of type τ and proceeds as type $\mathtt{A}\{M/x\}$, whilst a session of type $\forall x{:}\tau.\mathtt{A}$ 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:

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

denotes a session that will input an integer n[2] and 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 run time due to the existence of these proof objects, making this dependently-typed session framework a *de facto* model of proof-carrying code.

4.2. 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 run time. However, it is often the case that we want the specified properties to hold but we do not want to exchange the proof objects. Indeed, omitting the communication of proof objects may be sensible when the communicating parties have established trust by some external means, or when the properties are easily decidable and the proof objects can be *synthesised* by a decision procedure. For instance, in the example above, it is straightforward to check that the communicated numbers are indeed strictly positive.

To model the possibility of omitting proofs at run time, the work of [75, 89] extends the framework by internalising in the proof object language the concept of *proof irrelevance* [74], through a modality denoted $[\tau]$, which types terms of type τ that can be *safely* erased at run time. 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:

$$\mathtt{UpInterfaceI}(x) \triangleq x : \forall n{:}\mathtt{int}.\, \forall p{:}[n > 0].\, \exists y{:}\mathtt{int}.\, \exists q{:}[y > 0].\, \mathbf{1} \qquad (2)$$

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 run time and therefore can be safely omitted. This process of erasing proofs at run time is done in two steps: first all instances of proof irrelevant types and terms are replaced with the unit type and element (denoted $\mathtt{unit}$ and $\langle\rangle$, respectively). Since this procedure does not remove the communication step where the proof objects were previously exchanged, we may exploit the type isomorphisms,

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

[2]The different font for integers (n and n) is due to our convention that distinguishes between variables and values.

to consistently remove the communication overhead. 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 (sum types) which are inhabited by pairs. For example, we can rewrite `UpInterfaceI` as:

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

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 $\{x{:}\texttt{unit} \mid \sigma\}$, 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.

4.3. Affirmation and Digital Certificates

The examples above showcase what can be seen as two extremes in a spectrum of *trust*. In the type given in (1) no trust between the parties is assumed and therefore all proof objects must be made explicit in communication at run time. On the other hand, proof irrelevance as represented by the type given in (2) models a scenario of *full trust*, where no proof objects are expected at run time. In practice, there are tradeoffs 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, [75] also incorporates in the framework a notion of *affirmation* (from modal logic) of propositions and proofs by principals. Such affirmations can be realised 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 $\Diamond_{\mathsf{K}}\tau$ to the framework, which types objects that assert 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 necessarily trust the signing principal K to accept τ as true.

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

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

which expresses the type of a service that first inputs a function f, then accepts a verifier's word that each natural number is a prefixed point of f, and finally returns a fixed point of f to its client. Observe that object p is a *certificate* of the fact that f satisfies this property. In realistic scenarios, such as proof-carrying file systems [45], the use of affirmation and proof irrelevance results in substantially less communication overheads when compared to proof-carrying code in the sense of [70], where the proof objects become too big to be transmitted and checked every time a file is accessed.

5. Secure Interactions with Untrusted Components

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 those 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 environment.

A second, more general scenario 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 ensure that all compliant participants share an identical view of a session execution?

5.1. A Secure Protocol Compiler

The works [28, 29, 30] offer a first answer to these questions (the journal paper [30] merges and extends both [29] and [28]). The proposed language, expressed as a type language with a global graph-like representation (called session graphs), includes messages, roles, and sessions; it does not support parallelism or asynchrony. A secure implementability condition, defined on session graphs, is identified.

The principle of [30] is to use the session graph 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 graph 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 is defined through the notion of *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 in [30] 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. This approach is implemented as an extension of OCaml. A compiler has been developed, which takes as input a session description and produces as output an OCaml module with a function for each participant. Any user code calling one of these functions is guaranteed through the OCaml 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. As a case study, the authors of [30] implement and evaluate a conference management system with three roles: the program committee, an author, and a submission manager.

Two different extensions of the approach in [30] are developed in [76] and [14]. The work [76] generalises [30] with a more abstract setup and a greater session expressiveness. Rather than a graph representation, the authors of [76] define session specifications with CCS-like processes, which represent the desired communication pattern. A history-tracking process calculus is used as the lower-level model of a secure implementation. The correctness of the history-tracking mechanism with respect to the CCS specification is proved using a trace-based semantics which adequately models an adversary that can control the network and remote peers. This captures and generalises the signature-based mechanism of [30]. The work [14] improves [28, 30] with simpler and more efficient cryptography (using a combination of asymmetric and symmetric cryptography), and extends the session description language with value annotations (i.e., constraints on payload). This extension allows one to model commitments and to protect the integrity of each payload. As in [28, 30], 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.

5.2. Contract-oriented Service Composition

In the design of session-typed distributed applications according to a top-down approach [61], a choreography describing the global interaction behaviour of the application is projected to a set of local types, which describe the contributions of individual participants in the application. Each participant is implemented by a concrete process: if all these implementations respect their local types, the overall application is guaranteed to enjoy some correctness property (e.g., the absence of deadlocks).

In an adversarial setting, however, one cannot assume that the implementation of an untrusted participant respects its local type; indeed, participants have full control of the code they run, and they can even change it at run time. Any static analysis that requires inspection of the code of each participant is then pointless: consequently, properties which are not enforceable by run-time monitoring (e.g., the absence of deadlocks) cannot be enforced at all in this adversarial setting.

To cope with this situation, a different design approach has been proposed where the composition of distributed components is performed in a bottom-up fashion. In this approach, participants first advertise their promised behaviour as *contracts* to some broker; the broker inspects such contracts, and creates sessions among participants whose contracts admit an *agreement*. For instance, contracts could be binary or multiparty session types, and agreement could be one of the compliance / compatibility relations defined over them [62]. Once these sessions are created, participants can perform the actions prescribed by their contracts (in the case where they are session types, this would result in performing the prescribed inputs and outputs). An execution monitor can then keep track of the state of each contract with respect to the participants' actions, which cannot depart from the actions expected by the advertised contracts;

furthermore, the monitor can establish who is *accountable* at each step, i.e., responsible for the next interaction. Systems developed under this design approach are called *contract-oriented systems* [11].

While untrusted participants may cause deadlocks in both contract-oriented and top-down approaches, an advantage of the former is that one can use contracts to single out the participants which have breached the agreement, thus causing the deadlock. This can be associated with sanctions imposed to the culpable participants; these sanctions could range, e.g., from lowering the participants' reputation or imposing them a fine, to removing them altogether from the repository of available services.

5.3. Honesty in Contract-oriented Systems

Interacting with ill-typed or untrusted participants may have non-obvious consequences, especially in the case of applications whose implementations involve multiple interleaved sessions. For instance, consider a simple e-commerce scenario involving three participants: a seller A, a wholesaler B, and a client C that interact through binary sessions s (between A and B) and t (between A and C). Suppose that A is expected to receive a payment from C (in session t), then order an item to B (in session s), wait until the item is received (still in s), and finally ship the item to C (in t). If the wholesaler B receives the payment but never sends the item to A, then A becomes unable to ship the item to C. In turn, C may get stuck and be unable to advance in other sessions. Clearly, the problem may cascade to affect other sessions and participants.

A desirable goal for the designer of the seller would be to guarantee that A is never responsible for some stuck session: therefore, even if s is blocked by B, participant A will still behave according to his contract in t. This property — called *honesty* — is formally defined and investigated in the contract-oriented specification language CO_2 [12]. Honesty can be seen as multi-session well-typedness: if a participant is honest, his process implementation will behave according to his contracts in each session he establishes, even if other participants will not cooperate.

In general, a developer would aim at publishing only honest services that always respect contracts — even when the other participants are malicious: otherwise, the service infrastructure may eventually sanction him for contract breaches. Since honesty cannot be enforced by run-time monitoring (it is a sort of deadlock-freedom property), static analysis techniques for detecting honesty of processes are required. While honesty is not decidable in general [12], it can be statically approximated: as usual, the approximation must stay "on the safe side", i.e., if it statically determines that a service is honest, then this is really the case; otherwise, it may be either the case that the service is honest or it is not. In the literature, analysis techniques for honesty have been proposed using both type systems [10] and model checking [9].

5.4. Protection Against Untrusted Brokers

In contract-oriented applications, participants advertise their contracts to some broker, which establishes sessions among participants whose contracts admit an agreement. In such a scenario, the agreement property guarantees that — even in the presence of malicious participants — no interaction driven by the contracts will ever go wrong: in the worst case, if some participant does not reach his objectives, then some other (dishonest) participant will be culpable of a contract infringement.

In the above workflow, it is often assumed that brokers are trusted, in that they never establish a session in the absence of an agreement. In more byzantine scenarios, it may happen that a fraudulent broker creates a session where participants interact in the absence of an agreement. In this way, the broker may allow an accomplice to swindle an unaware participant. Note that the accomplice may perform his scam while adhering to his contract, and so he cannot be blamed for violations. A crucial problem is how to devise contracts which protect participants from malicious brokers. In contexts where brokers are malicious, contracts should still allow participants to reach their goals when the other participants are cooperative. At the same time, contracts should prevent participants from performing imprudent actions which could be exploited by malicious participants.

This problem has been addressed in [7, 8] in a game-theoretic setting, where session interactions are interpreted as games over *event structures* (ESs [94]), and participants are the players of these games. In this setting, a participant wins in a *play* (a trace of the ES) when he reaches success, or some other participants can be blamed for a violation. The idea is that the infrastructure will eventually inflict sanctions to the participants who have violated their contracts, as in [69].

Two key notions in this model are those of *agreement* and *protection*. Agreement is a property of contracts which guarantees that each honest participant may reach success whenever the other participants cooperate. Moreover, if an honest participant does not reach success, then some other participant can be blamed. A contract *protects* its participant if, whenever composed with any other contract (possibly that of an adversary), the contract admits at least one *non-losing strategy*, i.e., a strategy that guarantees that the participant will never end up in a failure state.

The notion of agreement in the game-based model is related in [7] to the progress-based notion of compliance in session types [5]. More precisely, two session types are compliant if and only if, in their interpretation as ESs, *all* (innocent) strategies are winning. Hence, compliance implies agreement, while the converse does not hold. This is illustrated by the following example, using the syntax of Section 2. Consider the following session types (where labels are to be viewed as branching labels):

$$\mathtt{T}_1 = !\mathtt{A}.!\mathtt{C}.\mathtt{end} \oplus !\mathtt{B}.\mathtt{end} \qquad \text{and} \qquad \mathtt{T}_2 = ?\mathtt{A}.\mathtt{end} + ?\mathtt{B}.\mathtt{end}.$$

where $\mathtt{T}_1$ is advertised by Player 1, and $\mathtt{T}_2$ by Player 2. We have that Player 2 agrees with the composition of $\mathtt{T}_1$ and $\mathtt{T}_2$. Indeed, the only innocent strategy for Player 2 is the one which prescribes him to:

- do $?\mathtt{A}$ after $\mathtt{T}_1$ has performed $!\mathtt{A}$;
- do $?\mathtt{B}$ after $\mathtt{T}_1$ has performed $!\mathtt{B}$.

This strategy is winning, because it leads Player 2 to the success state `end` in both cases (unless Player 1 violates $\mathtt{T}_1$). Similarly, Player 1 *agrees* with the composition of $\mathtt{T}_1$ and $\mathtt{T}_2$: his winning strategy is just to choose the branch $!\mathtt{B}$. However, the session types $\mathtt{T}_1$ and $\mathtt{T}_2$ are *not* compliant according to [5]. Indeed, if $\mathtt{T}_1$ takes the internal choice $!\mathtt{A}$, then a deadlock state is reached.

If brokers are dishonest, then they may establish sessions even in the absence of an agreement. For instance, assume Player 3 advertises the session type:

$$\mathtt{T}_3 = !\mathtt{PAY}.?\mathtt{RECEIVE}.\mathtt{end} \oplus !\mathtt{ABORT}.\mathtt{end}$$

A dishonest broker could make Player 3 interact with another player with session type:

$$\mathtt{T}_4 = ?\mathtt{PAY}.\mathtt{end}$$

Note that Player 3 does *not* agree with the composition of $\mathtt{T}_3$ and $\mathtt{T}_4$, because if his strategy chooses $!\mathtt{PAY}$, then he will lose if the other player does not perform $!\mathtt{RECEIVE}$ (as in $\mathtt{T}_4$), while if it chooses $!\mathtt{ABORT}$, then he will reach a *tie* state (i.e., neither success nor failure). However, Player 3 can protect himself against such dishonest brokers: a strategy that protects him would be the one which only chooses $!\mathtt{ABORT}$ (intuitively, doing $!\mathtt{PAY}$ leads Player 3 to lose if the other player never does $!\mathtt{RECEIVE}$, while doing $!\mathtt{ABORT}$ leads Player 3 to a tie state independently of the contract and of the strategy of the other player).

While, in general, it is not always possible to guarantee that a set of contracts admit both agreement and protection (as proved in [7]), it is possible to reconcile these two notions by relaxing the classical notion of causality, i.e., by assuming that some events can occur in the absence of a causal justification in the *past*, provided they have a justification in the *future* [8].

6. Emerging Directions

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

This situation is aggravated 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 requirements or contextual conditions.

While legacy code maintenance will always be an issue and integration is of utmost concern, a more recent trend focuses on anticipating how a system can react to changes in its external environment or requirements. Since types may provide specifications of correct behaviour, adaptation at run time may be guided by type information.

In the remainder of this section we describe some dynamic typing analysis approaches that cope with partial (static) type information and type-driven adaptation. We also briefly mention a connection between techniques for ensuring access control and behavioural typing, in particular considering the importance of *roles* in communication-centred systems.

6.1. Gradual Typing and Security

One approach to introduce behavioural types into existing systems is to do it gradually, on a per-module basis, so that typed and less typed program parts must interoperate. Gradual typing [82, 83] addresses exactly this interoperation. Gradual type systems have been developed from dynamic type systems [1, 54]. They provide a type `Dynamic` with operations to *inject* a value of arbitrary type into `Dynamic` as well as operations to *project* a dynamic value to an arbitrary type. While an injection always succeeds, a projection may fail and throw an exception if the statically expected type does not agree with the run-time type of the dynamic value. For example, the injection for type `int` maps 1 of type `int` to the pair $\langle \texttt{int}, 1 \rangle$, which represents the dynamic value. The corresponding projection checks the type in the first component against the expected type. It returns the second component if the types match and throws an exception, otherwise. These operations can enhance a conventionally typed language [1] or they can form the basis for optimising a dynamically typed language [54].

Gradual typing is also applicable in a security context [38]. The pure λ-calculus setting of the first approach has later been extended to an ML core language. This extension employs a very liberal treatment of memory references that are shared between statically and dynamically typed fragments [41].

Most security type systems that control information flow and track data integrity assume an underlying program that is well typed according to some standard type system [93]. 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, as already discussed.

For simplicity, existing work on gradual security typing [38, 41] assumes an underlying typed program and restricts the gradual aspect of the system to the security labels. Gradual security typing guarantees termination insensitive noninterference [50]. The statically security-typed parts observe this property by means of the type system, whereas the dynamically security-typed parts observe the property with a monitor that enforces the no-sensitive-upgrade policy on run-time security labels.

Consider the example of a function `f(s,x)` that manages information flow with a low-security boolean argument `s` that indicates the security level of `x`, which comes with a dynamic security level. A gradual security system would annotate the function as follows.

```
f(low s, dyn x) {
  if(!s)
    publish_low(x : dyn => low);
}
```

The function `publish_low` takes a low-security argument and writes it to a public channel. The function `f` passes `x` to it after applying the coercion from `dyn` to `low` security. This coercion fails if `s` does not indicate the security level of `x` correctly.

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 memory references, it restricts the use of references that are shared between statically and dynamically typed parts of a program. For that reason, the

language proposed in [41] requires some dynamic checks even in the statically typed parts of a program.

Subsequent work on gradual annotated types [42] indicates that the execution model for gradual security with references can be improved to the point that statically typed parts need no dynamic checks. Ongoing work considers the formalisation and implementation of a system improved along these lines in the context of a Java-like language.

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 places security coercions at the borders of the legacy code, potentially adding run-time labels to all values, and monitors its execution. In contrast, new code would run without labels at full speed because its security properties are guaranteed by the static type system.

6.2. Gradual Security Typing and Sessions

Addressing the connection of legacy code with new code via communication channels is the point where session types or other behavioural types enter the scene. The first behavioural type system that was extended with gradual features was the typestate system of the concurrent object-oriented language Plaid [85, 95]. A typestate system keeps track of the current state of an object statically. An example is the typestate of a file object that reflects whether the file is open or closed in its type. As the typestate changes when operations are applied to the file, it must be linear or affine, just like the channel type in a session type system. Here is a very simple file API with typestate:

$$\begin{aligned} \texttt{open} &: \texttt{filename} \to \texttt{file}[\textsc{open}] \\ \texttt{readInt} &: \texttt{file}[\textsc{open}] \to \texttt{int} \\ \texttt{close} &: \texttt{file}[\textsc{open} \rightsquigarrow \textsc{closed}] \to \texttt{unit} \end{aligned}$$

The notation $\textsc{open} \rightsquigarrow \textsc{closed}$ indicates that a file in state OPEN is expected and that it evolves to state CLOSED on executing the function.

The coercion of a value with typestate into the dynamic type reifies its current typestate in a run-time value. Operations on the dynamic type step through an automaton that executes the same transitions as the static typestate computation, similar to communicating automata [32]. The authors of the gradual typestate work suggest to use the dynamic type during program development because their static typestate system requires program annotations to manage situations where there is more than one memory reference to the same object (i.e., aliasing).

The work on gradual types for Plaid cannot be transferred readily to session types. The obstacle lies in dealing with the linearity of the channel types, where Plaid resorts to (sophisticated) alias management. Fortunately, it has been shown that linearity and gradual typing are largely orthogonal and that many important results from standard gradual typing carry over to a setting with linear types [40]. For affine types, it is possible to gradualise the affine property [90]. It is unlikely that a similar gradualisation can be achieved for linear types.

One important result in gradual typing is the blame theorem [88]. It strengthens the progress property of a type system by making precise that failing coercions can always be attributed to the less precisely typed side of the coercion. In this context,

progress means that a well-typed term is either a value, or it can perform an evaluation step, or it fails at a cast to a more precise type. The blame theorem clearly locates the demarcation between statically proved and dynamically checked code at a particular kind of coercions that coerce from static to dynamic.

Given these preliminaries, 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., [22, 23, 24]) can form the basis of a gradual system, as in the setting without sessions. First steps towards integrating gradual typing with session types have been taken [87].

6.3. Run-time Adaptation

As (communication-centred) software systems rely on highly dynamic infrastructures, such as those built on cloud-based platforms, the ability of adapting to varying requirements and external conditions becomes crucial to ensure uninterrupted, correct system behaviour. There is a bidirectional relation between run-time adaptation and security requirements:

(a) On the one hand, it is plausible to react to security threats by executing an adaptation routine that, e.g., replaces/updates the affected component;

(b) On the other hand, one would like adaptation mechanisms which address general functional requirements but also preserve established security policies. We would like to avoid, e.g., mechanisms that update faulty components with correct but insecure patches.

In the light of this interplay between security and run-time adaptation, a comprehensive approach that exploits their relationships appears natural. This is the main motivation of the paper [25], which integrates security guarantees (access control and secure information flow) with self-adaptation in a process framework of multiparty structured communications, exploring the above relation (a). More precisely, behavioural types with security levels are used to monitor reading and writing violations, corresponding to access control violation and information leaks, respectively. Behavioural types define security policies by stipulating read and write permissions, represented by security *levels*. While a read permission is an upper bound for the level of incoming messages, a write permission is a lower bound for the level of outgoing messages. Accordingly, a reading or writing violation occurs when a participant attempts to read or write a message whose level is not allowed by the corresponding read or write permission. An associated operational semantics is instrumented so as to trigger adaptation mechanisms in case of violations, but also to prevent the violations from occurring and propagating their effect in the choreography.

The framework of [25] consists of a language for processes and networks, global types, and run-time monitors. Run-time monitors are obtained as projections from global types onto individual participants. This way, behavioural types provide a clear description for enforcing dynamic monitoring of participants. Processes represent code that will be coupled with monitors to implement participants. A network is a collection of monitored processes which realise a choreography as described by the global type.

The semantics of networks includes both *local* and *global* adaptation mechanisms; their goal is to handle minor and serious violations, respectively.

Informally, the local adaptation mechanism "ignores" unauthorized actions and modifies a monitored process at run time; it relies on a *collection* of typed processes, which contains all processes which may be used in reconfiguration steps. In case of a read violation, the local adaptation mechanism modifies the behaviour of the monitor so as to omit the disallowed read, and then injects a process from the collection that is compliant with the new monitor. In case of a write violation, the local adaptation mechanism penalises the sender by decreasing the read level of his monitor and replacing the implementation for the receiver. Here again the new implementation is extracted from the collection of typed processes. Therefore, adaptation is local insofar as reconfiguration steps concern only one monitored process.

The global mechanism for adaptation relies on distinguished low-level values called *nonces*. When an attempt to leak a value is detected, a freshly generated nonce is passed instead. This mechanism has two goals: first, to avoid improperly communicating the protected value; second, to allow the whole system to make progress, for the benefit of the participants not involved in the violation. At any point, the semantics may trigger a reconfiguration routine that replaces the portion of the choreography involving the participants that may propagate a nonce. Thus, in the global adaptation mechanism, a part of the choreography is isolated and replaced, preserving the correctness of the whole system. Notice that the function which returns a new choreography given a choreography with nonces is left unspecified in [25]; this function is intended as a parameter of the operational semantics.

To illustrate the kind of scenarios that the framework in [25] aims to target (but also the intuition underlying minor and serious violations), consider a choreography involving a user, his bank, a store, and a social network. Exchanges occur on top of a browser, which relies on plug-ins to integrate information from different services. Agreed exchanges between the user, the bank, and the store may in some cases lead to a (public) message announcement in the social network. One would like to ensure that the buying protocol works as expected, but also to avoid that sensitive information, exchanged in certain parts of the protocol, is leaked. Such an undesired behaviour should be corrected as soon as possible. In fact, one would like to stop relying on the (unreliable) participant in ongoing/future instances of the protocol. Depending on how serious the violation is, however, one may also like to react in different ways.

- If the leak is minor (e.g., because the user interacted incorrectly with the browser), one may then simply identify the source of the leak and postpone the reaction to a later stage, enabling unrelated participants in the choreography to proceed with their exchanges.

- Otherwise, if the leak is serious, one may then wish to adapt the choreography as soon as possible, removing the plug-in and modifying the behaviour of the involved participants. This form of reconfiguration, however, should only concern the participants involved with the insecure plug-in; participants not directly affected by the leak should not be unnecessarily restarted. In this simple example, since the unintended social announcement concerns only the user, the store and the social network, updates should not affect the behaviour of the bank.

Notice that, for simplicity, the framework in [25] non-deterministically selects between local and global mechanisms. This aspect can be refined, considering that the actual meaning of minor and serious violations often depends on the applications at hand. Finally, in the light of the bidirectional relation between adaptation and security mentioned above, it is clear that the framework in [25] follows direction (a) (i.e., it is a form of security-driven adaptation); addressing direction (b) (i.e., forms of correctness-driven adaptation that preserve security policies) is an interesting topic for future work.

6.4. Role-based Access Control

In an open distributed network, it is extremely important to provide security and protect privacy during transfer and management of data. In a series of papers [35, 36, 37, 47, 63], behavioural types for distributed systems containing semi-structured XML data are investigated. These works introduce type-based verification techniques using the model presented in [44] (where the focus is on the model, including its behavioural theory): a network of peers is a parallel composition of locations, where each location consists of a data tree and a process. Locations are enriched by security policies prescribing how the access permissions of roles can be modified. Processes have roles and edges in data trees are associated with roles, representing permissions (to access edges) assigned to roles. Well-typed systems respect prescribed security policies and role-based access control.

The issues addressed by the above mentioned works, namely role-based access control [35], are also a main concern when focussing on communication-centred systems, since the notion of role may also be involved, in particular in the context of security protocols. More recently, roles have also gained a behavioural connotation, as communicating parties may impersonate different roles throughout their execution and roles may actually be carried out by several parties, in particular when *delegation* is involved. Delegation is a challenging feature from a security perspective since it involves yielding access of a (potentially secure) medium to third parties. In [48] a first step in characterising the delegation of *authorisations* to impersonate roles is taken, building on the type-based verification techniques introduced in the approaches mentioned above, in particular [35], and on the behavioural type system given in [3]. Resources are structured, in some sense, in a way which is similar to that imposed by behavioural types. Thus, access control for structured resources paves the way to access control for *communication* resources in a structured protocol of interaction.

7. Conclusions

Security and trustworthiness are essential properties for software systems. In the context of distributed applications, the challenge of enforcing these properties is tied to the consistency of structured conversations among parties. In fact, since exchanges of (sensitive) data in such applications often follow predefined communication sequences, security properties go hand in hand with safety and liveness properties associated to correct protocols, such as conformance/compliance, resource usage, and deadlock-freedom/progress. As a consequence, the integration of techniques for describing and enforcing both kinds of properties is indispensable in many settings. This

	Static			Dynamic	
	Enhanced BTs	Contracts	Extended DTs	Gradual	Adaptive
AC	Section 3.1 Section 6.4		Section 3.1		
SIF	Section 3.2			Section 6.1	
Integrity	Section 3.3 Section 4 Section 5.1	Section 5	Section 4		Section 6.3
Safety	Section 3.1 Section 3.2 Section 3.3			Section 6.2	

Table 1: Summary of proposals.

paper presents an overview of efforts to achieve this integration in a rigorous way, building upon calculi and models for communicating processes. We focus on work based on *behavioural types*, which extend the well-established concept of data types to describe complex communication structures.

Our review illustrates how the integration of security concerns into approaches based on behavioural types leads to a rich landscape of models and techniques, with both foundational and practical significance.

On the foundational side, the overview starts with extended models of session-based communication, which cover a wide variety of security-related concerns, including access control, secure information flow, and data integrity. A fruitful research strand concerns logic-based approaches to behavioural types. Approaches based on linear logic lead to clean, extensible typed models in which notions of resource-awareness and trustworthy communication have principled justifications. In particular, aspects such as proof-carrying code and digital certificates can be integrated in a session-typed setting by building upon appropriate (linear) logical grounds. From a more practical perspective, we examine ways to reconcile the usual assumptions of typed models with the actual requirements of distributed communications over open networks. These efforts concern the development of compilers of protocols with cryptographic information, but also models of honest, contract-based communication (in which service agreements are handled bottom-up by a broker), and theories of protection for contracts, which aim at ensuring honest participants and trusted brokers. We also discuss models of gradual typing, in which the combination of static and dynamic types turns out to be useful to integrate parts of the system not amenable to static typing (such as legacy code) and to account for dynamic security policies. Behavioural types can be used to monitor system execution in case of security violations, and to guide adaptations which prevent such violations to occur.

Table 1 summarises the proposals described in this paper, organised in two dimensions. The first dimension, organised in columns, concerns the kind of approach considered: Enhancements of BTs (Enhanced BTs), Contracts, Extended Datatypes (Extended DTs), Gradual and Adaptive Types. The second dimension, represented by rows, concerns the targeted security properties: Access Control (AC), Secure Informa-

tion Flow (SIF), Integrity, and Safety. It is immediate to see that enhancing behavioural types (and the model/framework in which they are considered) is the most common approach to address security properties. On a somewhat orthogonal perspective, we may also observe that several approaches address integrity concerns so as to transport security properties into untrusted environments.

In our view, strengthening security guarantees via enhanced type disciplines and the transfer of such guarantees to less controlled (more realistic) environments constitute two important axes along which forthcoming research in behavioural types for security should proceed. In particular, we believe that the consolidation of techniques that relate specifications to implementations is a crucial building block for more reliable and secure applications. A considerable body of work has been carried out in the last years on ensuring "safe" protocols by focusing on protocol specifications, since implementations are too low-level to support reasoning on protocol properties. By ensuring that implementations conform to specifications in a rigorous way, the properties established for the specification may carry over to actual implementations.

For the sort of complex security properties that we have discussed in this document, we believe that lifting the analysis to the level of specifications/types enables reasoning at a more adequate abstraction level. In particular, since it is crucial to consider how such properties may be transferred to open environments, we need to focus on techniques that support local reasoning, building upon clearly defined compositionality principles. Indeed, compositionality is the key to enable scalable and tractable analyses.

From a practical perspective, an opportunity for future research concerns integration of models of security monitoring into emerging practical languages and frameworks based on behavioural types. For instance, the Scribble protocol language [96] provides support for developing large-scale distributed applications whose interaction architecture can be expressed as multiparty session types. Scribble has been used for implementing run-time type checking (monitoring) of communicating processes [15, 26], also in collaboration with the Ocean Observatories Initiative [71].

Acknowledgments *We acknowledge support from COST Action IC1201* Behavioural Types for Reliable Large-Scale Software Systems *(BETTY) and we thank the members of BETTY working group on Security (WG2) for interesting related discussions. We are grateful to the anonymous reviewers for their feedback and many insightful remarks, which greatly helped us to improve the quality of this document.*

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Prog. Lang. Syst.*, 13(2):237–268, 1991.

[2] D. Ancona, V. Bono, M. Bravetti, G. Castagna, J. Campos, P.-M. Deniélou, S. Gay, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. Martins, V. Mascardi, F. Montesi, N. Ng, R. Neykova, L. Padovani, V. Vasconcelos, and N. Yoshida. Behavioral types in programming languages. `http://www.behavioural-types.eu/publications/WG3-State-of-the-Art.pdf`, 2014.

[3] P. Baltazar, L. Caires, V. T. Vasconcelos, and H. T. Vieira. A type system for flexible role assignment in multiparty communicating systems. In *TGC 2012*, volume 8191 of *LNCS*, pages 82–96. Springer, 2013.

[4] P. Baltazar, D. Mostrous, and V. T. Vasconcelos. Linearly refined session types. In *LINEARITY 2012*, volume 101 of *EPTCS*, pages 38–49, 2012.

[5] F. Barbanera and U. de'Liguoro. Two notions of sub-behaviour for session-based client/server systems. In *Proc. PPDP*, pages 155–164, 2010.

[6] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[7] M. Bartoletti, T. Cimoli, G. M. Pinna, and R. Zunino. Contracts as games on event structures. J. Log. Alg. Meth. Prog. (2015), `http://dx.doi.org/10.1016/j.jlamp.2015.05.001`, in press.

[8] M. Bartoletti, T. Cimoli, and R. Zunino. A theory of agreements and protection. In *POST 2013*, volume 7796 of *LNCS*, pages 186–205. Springer, 2013.

[9] M. Bartoletti, M. Murgia, A. Scalas, and R. Zunino. Modelling and verifying contract-oriented systems in Maude. In *WRLA 2014*, volume 8663 of *LNCS*, pages 130–146. Springer, 2014.

[10] M. Bartoletti, A. Scalas, E. Tuosto, and R. Zunino. Honesty by typing. In *FMOODS/FORTE 2013*, volume 7892 of *LNCS*, pages 305–320. Springer, 2013.

[11] M. Bartoletti, E. Tuosto, and R. Zunino. Contract-oriented computing in CO2. *Sci. Ann. Comp. Sci.*, 22(1):5–60, 2012.

[12] M. Bartoletti, E. Tuosto, and R. Zunino. On the realizability of contracts in dishonest systems. In *COORDINATION 2012*, volume 7274 of *LNCS*, pages 245–260. Springer, 2012.

[13] L. Bettini, M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR 2008*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.

[14] K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF 2009*, pages 124–140. IEEE, 2009.

[15] L. Bocchi, T. Chen, R. Demangeon, K. Honda, and N. Yoshida. Monitoring networks through multiparty session types. In *FMOODS/FORTE 2013*, volume 7892 of *LNCS*, pages 50–65. Springer, 2013.

[16] L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR 2010*, volume 6269 of *LNCS*, pages 162–176. Springer, 2010.

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

[18] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. T. Vasconcelos, and G. Zavattaro. SCC: A service centered calculus. In *WS-FM 2006*, volume 4184 of *LNCS*, pages 38–57. Springer, 2006.

[19] A. Bossi, R. Focardi, C. Piazza, and S. Rossi. Verifying persistent security properties. *Comp. Lang., Syst. & Struct.*, 30(3-4):231–258, 2004.

[20] R. Bruni and L. G. Mezzina. Types and deadlock freedom in a calculus of services, sessions and pipelines. In *AMAST 2008*, volume 5140 of *LNCS*, pages 100–115. Springer, 2008.

[21] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR 2010*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.

[22] S. Capecchi, I. Castellani, and M. Dezani-Ciancaglini. Typing access control and secure information flow in sessions. *Inf. Comput.*, 238:68–105, 2014.

[23] S. Capecchi, I. Castellani, and M. Dezani-Ciancaglini. Information flow safety in multiparty sessions. *Math. Struct. in Comp. Science* (2015), `http://journals.cambridge.org/abstract_S0960129514000619`.

[24] S. Capecchi, I. Castellani, M. Dezani-Ciancaglini, and T. Rezk. Session types for access and information flow control. In *CONCUR 2010*, volume 6269 of *LNCS*, pages 237–252. Springer, 2010.

[25] I. Castellani, M. Dezani-Ciancaglini, and J. A. Pérez. Self-adaptation and secure information flow in multiparty structured communications: A unified perspective. In *BEAT 2014*, volume 162 of *EPTCS*, pages 9–18, 2014.

[26] T.-C. Chen, L. Bocchi, P.-M. Deniélou, K. Honda, and N. Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *TGC 2011*, volume 7173 of *LNCS*, pages 25–45. Springer, 2012.

[27] S. Chong, A. C. Myers, K. Vikram, and L. Zheng. *Jif Reference Manual*. Cornell University, 2009. `http://www.cs.cornell.edu/jif`.

[28] R. Corin and P.-M. Deniélou. A protocol compiler for secure sessions in ML. In *TGC 2007*, volume 4912 of *LNCS*, pages 276–293. Springer, 2007.

[29] R. Corin, P.-M. Deniélou, C. Fournet, K. Bhargavan, and J. J. Leifer. Secure implementations for typed session abstractions. In *CSF 2007*, pages 170–186. IEEE, 2007.

[30] R. Corin, P.-M. Deniélou, C. Fournet, K. Bhargavan, and J. J. Leifer. A secure compiler for session abstractions. *J. Comp. Sec.*, 16(5):573–636, 2008.

[31] S. Crafa and S. Rossi. A theory of noninterference for the pi-calculus. In *TGC 2005*, volume 3705 of *LNCS*, pages 2–18. Springer, 2005.

[32] P. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP 2012*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.

[33] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[34] M. Dezani-Ciancaglini, S. Drossopoulou, D. Mostrous, and N. Yoshida. Objects and session types. *Inf. Comput.*, 207(5):595–641, 2009.

[35] M. Dezani-Ciancaglini, S. Ghilezan, S. Jaksic, and J. Pantovic. Types for role-based access control of dynamic web data. In *WFLP 2010*, volume 6559 of *LNCS*, pages 1–29. Springer, 2010.

[36] M. Dezani-Ciancaglini, S. Ghilezan, and J. Pantovic. Security types for dynamic web data. In *TGC 2006*, volume 4661 of *LNCS*, pages 263–280. Springer, 2006.

[37] M. Dezani-Ciancaglini, S. Ghilezan, J. Pantovic, and D. Varacca. Security types for dynamic web data. *Theor. Comput. Sci.*, 402(2-3):156–171, 2008.

[38] T. Disney and C. Flanagan. Gradual information flow typing. In *STOP 2011*, 2011.

[39] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity OS. In *EuroSys 2006*, pages 177–190. ACM, 2006.

[40] L. Fennell and P. Thiemann. The blame theorem for a linear lambda calculus with type dynamic. In *TFP 2012*, volume 7829 of *LNCS*, pages 37–52. Springer, 2012.

[41] L. Fennell and P. Thiemann. Gradual security typing with references. In *CSF 2013*, pages 224–239. IEEE, 2013.

[42] L. Fennell and P. Thiemann. Gradual typing for annotated type systems. In *ESOP 2014*, volume 8410 of *LNCS*, pages 47–66. Springer, 2014.

[43] R. Focardi and R. Gorrieri, editors. *Foundations of Security Analysis and Design, Tutorial Lectures*, volume 2171 of *LNCS*. Springer, 2001.

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

[45] D. Garg and F. Pfenning. A proof-carrying file system. In *S&P 2010*, pages 349–364. IEEE, 2010.

[46] S. J. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2-3):191–225, 2005.

[47] S. Ghilezan, S. Jaksic, J. Pantovic, and M. Dezani-Ciancaglini. Types and roles for web security. *Trans. Adv. Res.*, 8(2):16–21, 2012.

[48] S. Ghilezan, S. Jaksic, J. Pantovic, J. A. Pérez, and H. T. Vieira. Dynamic role authorization in multiparty conversations. In *BEAT 2014*, volume 162 of *EPTCS*, pages 1–8, 2014.

[49] J.-Y. Girard. Linear Logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[50] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. Sec. and Priv.*, pages 11–20, 1982.

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

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

[53] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: tracking information flow in JavaScript and its APIs. In *SAC 2014*, pages 1663–1671. ACM, 2014.

[54] F. Henglein. Dynamic typing: Syntax and proof theory. *Sci. Comput. Programming*, 22:197–230, 1994.

[55] M. Hennessy. The security pi-calculus and non-interference. *J. Log. Algebr. Program.*, 63(1):3–34, 2005.

[56] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM Trans. Program. Lang. Syst.*, 24(5):566–591, 2002.

[57] K. Honda. Types for dyadic interaction. In *CONCUR 1993*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.

[58] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP 1998*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

[59] K. Honda, V. T. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *ESOP 2000*, volume 1782 of *LNCS*, pages 180–199. Springer, 2000.

[60] K. Honda and N. Yoshida. A uniform type structure for secure information flow. *ACM Trans. Program. Lang. Syst.*, 29(6), 2007.

[61] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL 2008*, pages 273–284. ACM, 2008.

[62] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro. Foundations of behavioural types. `http://www.behavioural-types.eu/publications/WG1-State-of-the-Art.pdf`, 2014.

[63] S. Jaksic. Input/output types for dynamic web data. In *ICTCS 2012*, 2012.

[64] D. Kitchin, A. Quark, W. R. Cook, and J. Misra. The Orc programming language. In *FMOODS/FORTE 2009*, volume 5522 of *LNCS*, pages 1–25. Springer, 2009.

[65] N. Kobayashi. A Type System for Lock-Free Processes. *Inf. Comput.*, 177:122–159, 2002.

[66] N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Inf.*, 42(4-5):291–347, 2005.

[67] M. Kolundzija. Security types for sessions and pipelines. In *WS-FM 2008*, volume 5387 of *LNCS*, pages 175–190. Springer, 2008.

[68] A. Lapadula, R. Pugliese, and F. Tiezzi. Regulating data exchange in service oriented applications. In *FSEN 2007*, volume 4767 of *LNCS*, pages 223–239. Springer, 2007.

[69] A. Mukhija, A. Dingwall-Smith, and D. Rosenblum. QoS-aware service composition in Dino. In *ECOWS*, pages 3–12, 2007.

[70] G. C. Necula. Proof-carrying code. In *POPL 1997*, pages 106–119. ACM, 1997.

[71] Ocean Observatories Initiative, 2010. `http://www.oceanleadership.org/programs-and-partnerships/ocean-observing/ooi/`.

[72] L. Padovani. Deadlock and lock freedom in the linear π-calculus. In *CSL-LICS 2014*, page 72. ACM, 2014.

[73] J. A. Pérez, L. Caires, F. Pfenning, and B. Toninho. Linear logical relations and observational equivalences for session-based concurrency. *Inf. Comput.*, 239:254–302, 2014.

[74] F. Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *LICS 2001*, pages 221–230. IEEE, 2001.

[75] F. Pfenning, L. Caires, and B. Toninho. Proof-carrying code in a session-typed process calculus. In *CPP 2011*, volume 7086 of *LNCS*, pages 21–36. Springer, 2011.

[76] J. Planul, R. Corin, and C. Fournet. Secure enforcement for global process specifications. In *CONCUR 2009*, volume 5710 of *LNCS*, pages 511–526. Springer, 2009.

[77] F. Pottier. A simple view of type-secure information flow in the p-calculus. In *CSFW 2002*, pages 320–330. IEEE, 2002.

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

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

[80] P. Y. A. Ryan and S. A. Schneider. Process algebra and non-interference. In *CSFW 1999*, pages 214–227. IEEE, 1999.

[81] J. F. Santos and T. Rezk. An information flow monitor-inlining compiler for securing a core of JavaScript. In *SEC 2014*, volume 428 of *IFIP Adv. ICT*, pages 278–292. Springer, 2014.

[82] J. Siek and W. Taha. Gradual typing for objects. In *ECOOP 2007*, volume 4609 of *LNCS*, pages 2–27. Springer, 2007.

[83] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

[84] V. Simonet. *The Flow Caml System: Documentation and users manual*. INRIA, 2003. `http://www.normalesup.org/~simonet/soft/flowcaml/`.

[85] J. Sunshine, K. Naden, S. Stork, J. Aldrich, and É. Tanter. First-class state change in Plaid. In *OOPSLA 2011*, pages 713–732. ACM, 2011.

[86] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013.

[87] P. Thiemann. Gradual typing for session types. In *TGC 2014*, volume 8902 of *LNCS*, pages 144–158. Springer, 2014.

[88] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. In *OOPSLA 2006*, pages 964–974. ACM, 2006.

[89] B. Toninho, L. Caires, and F. Pfenning. Dependent session types via intuitionistic linear type theory. In *PPDP 2011*, pages 161–172. ACM, 2011.

[90] J. A. Tov and R. Pucella. Stateful contracts for affine types. In *ESOP 2010*, volume 6012 of *LNCS*, pages 550–569. Springer, 2010.

[91] A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of software components using session types. *Fundam. Inform.*, 73(4):583–598, 2006.

[92] V. T. Vasconcelos, S. J. Gay, and A. Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.

[93] D. Volpano, C. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *J. Comp. Sec.*, 4(2-3):167–187, 1996.

[94] G. Winskel. Event structures. In *APN 1986*, volume 255 of *LNCS*, pages 325–392. Springer, 1986.

[95] R. Wolff, R. Garcia, É. Tanter, and J. Aldrich. Gradual typestate. In *ECOOP 2011*, volume 6813 of *LNCS*, pages 459–483. Springer, 2011.

[96] N. Yoshida, R. Hu, R. Neykova, and N. Ng. The Scribble protocol language. In *TGC 2013*, volume 8358 of *LNCS*, pages 22–41. Springer, 2013.