Combining behavioural types with security analysis

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

\textsuperscript{a} Dipartimento di Matematica e Informatica, University of Cagliari, Italy  
\textsuperscript{b} INRIA Sophia Antipolis, France  
\textsuperscript{c} Royal Holloway, University of London, UK  
\textsuperscript{d} Dipartimento di Informatica, Università di Torino, Italy  
\textsuperscript{e} Faculty of Technical Sciences, University of Novi Sad, Serbia  
\textsuperscript{f} CITI and Departamento de Informática, FCT, Universidade Nova de Lisboa, Portugal  
\textsuperscript{g} University of Freiburg, Germany  
\textsuperscript{h} Carnegie Mellon University, USA  
\textsuperscript{i} University of Groningen, The Netherlands  
\textsuperscript{j} IMT Institute for Advanced Studies Lucca, Italy

Abstract

Today’s software systems are highly interconnected and increasingly rely on communication to achieve their goals; due to their societal importance, security and trustworthiness are crucial aspects in the correct behaviour of these systems. Behavioural types represent a widely studied approach to the enforcement of correctness properties in distributed systems. This paper offers a unified overview of approaches 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 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 such systems are distributed over the network, and thus they heavily rely 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 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 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 range from the inability of an object to handle a method call (message not understood), to races (competition among concurrent programs for some shared resource) which may lead to inconsistent states or unexpected behaviours, to communication errors, caused by non-matching expectations of two communicating partners.

Originally conceived for operating systems, static and dynamic techniques for ensuring access control and secure information flow have been gradually introduced both into specification languages such as process calculi and into fully-fledged programming languages. In today’s open and highly distributed computing environment, excluding security flaws before programs are deployed is crucial. In the last two decades, spurred by the pioneering work of type systems targeting security properties have been proposed for a number of core calculi and programming languages. Type-based approaches allow focussing on disciplined systems and ensure that such systems conform to the prescribed security policies (e.g., access control or secure information flow).

Behavioural types (BTs) for communication-centred systems have been studied for a variety of calculi and languages since the late nineties. The distinguishing feature of BTs is that they specify a system’s behaviour from an observational perspective, focussing on system interaction points, or interfaces. An interface-based approach is particularly convenient to describe and analyse heterogeneous software assemblies in which individual computing artifacts are only available as communicating black-boxes and/or are developed using different programming languages. BTs are usually tied to process calculi, where the compositional character of type systems fits well with the algebraic description of processes. In particular, the compositionality at the level of processes and types offers a natural way of specifying settings in which the (typed) interacting systems are meant to be inserted in (typically untyped) external contexts.

BT-based approaches must be fine-tuned to deal with the tension between expressiveness and tractability. The design of behaviourally typed languages requires 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. That is, the key is obtaining a balance between expressiveness and feasibility of analysis.

In this respect, session types [52, 53] stand out as a particularly attractive instance of behavioural types. Session types allow interactions to be structured into basic units called sessions. Then individual interaction patterns are 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 [78] and object-oriented programming [31]), and addressing also lower-levels of application (namely operating system design [37] and middleware communication protocols [77]), 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. In particular, session-typed models focus on open-ended systems, where loosely coupled parties may synchronise to start a session on a specific (public) service, thereafter interacting on the private
(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 with given roles 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 roles in which communication actions are carried out, 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, in practice this scenario is too restrictive. In realistic distributed applications, the components are spread over 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. One may only assume one part of the application to be typed, and therefore trusted. The challenge is then to guarantee that the desired security property continues to hold when the trusted part interacts with the untrusted one.

In this document, we present a survey of existing BT-based approaches that are aimed at ensuring security properties. Our review does not only reflect several different views of secure and trustworthy communication-based systems; it 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, also exploiting reputation and runtime adaptation. Section 4 presents behavioural types expressing security policies for distributed/structured data and provenance for linked data. Section 5 shows how the logical foundation of session types – based on the correspondence with linear logic – can be fruitful also for security. In particular, dependent session types and dynamic spatial logic are applied to proof carrying code and digital certificates. To some extent, it is fair to say that the approaches described in Sections 2–5 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 6 reviews a compilation of session-like descriptions into implementations of cryptographic protocols that ensures that honest session participants are protected from external interference. Section 6 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 6.5 by means of game theoretic approaches. Section 7 presents recent developments that allow to reconcile legacy code and varying security policies by means of gradual types, and Section 8 gives some concluding remarks.
2. Preliminaries

In this section we informally present some key notions pertaining to session types, to allow a more informed reading of the document. The interested reader is referred to the recent survey \[56\] for further details on the foundations of behavioural types.

Session types characterise how communication channels are used by programs. Intuitively, much like the typing of a var: int tells us that var will be used to hold integer values, a type 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 $S_1 = !\text{int}.?\text{bool}.\text{end}$: it 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 the end denotes no further usage. A fundamental feature of session types is that they capture linear interactions, i.e., a channel can interact only in one way at any time — notice that this is essential to enable the description of the next stage of the protocol, as there is only one possible behavioural path.

A type such as $S_1$ above is inherently deterministic. Determinism in session typed linear interactions does not exclude the specification of alternative behaviours, via so-called branching and choice types. Alternatives are modelled by an interface “menu” (usually implemented via a set of available inputs, and typed using branching) and a corresponding selection (usually implemented via an output, and typed using choice). Alternatives are conveniently identified by labels. For example, the session type

$$S_2 = \text{number}?\text{int}!.\text{int}.\text{end} + \text{condition}?\text{bool}!.\text{end}$$

may be used to describe a process that is waiting for the selection of either one of its offered options, identified by labels number and condition. This process can be safely composed with a process willing to select 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 result that ensures typing is preserved under system evolution, entails an evolution also at the level of types.

Another notion central to session types is that of (session) delegation, which allows for other parties to gain access to an already established interaction. Realised via channel name 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 (which may proceed with the protocol, unaware of the exchanges at the server side). Carried types can thus be session types themselves; this way, e.g., session type

$$S_3 = ?(!\text{int}.\text{end}).\text{end}$$

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

Session types for binary “client-server” interaction as introduced in \[53\] capture the two-ended interaction via the type of one of the endpoints — which are sometimes distinguished by so-called polarities \[43\], 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 and conversely — a notion usually referred to as duality.
This way, the duals of session types $S_1$ and $S_3$ above are $\overline{S_1} = \text{?int.!bool.end}$ and $\overline{S_3} = \text{!(int.end).end}$, respectively.

More recently, following the approach introduced in [54], generalisations of session types that capture multiparty interaction have been put forward. Pairwise linear interaction is still ensured, via the use of dedicated intra-session channels (or session indexes [11]) which also support a more realistic asynchronous model of communication. The session type of a single participant is no longer enough to capture the entire interaction among all participants; to this end, global types are introduced, which specify the message exchanges among the various collaborating parties in a structured way.

As a global type explicitly identifies the involved participants, it is possible to extract the individual contributions via a projection function that singles out the actions pertaining to each identified participant — these contributions are commonly referred to as local types.

The next section reports on approaches that extend the original multiparty session types proposal so as to deal with security concerns.

Another session-based approach that addresses multiparty interaction has been introduced in [21], where labels are exploited to maintain pairwise linear (synchronous) interactions. The notion of “projection” introduced in [21] targets the identification of the contributions of the individual parties in a compositional way, one subsystem at a time, and is not driven by participant identities (which are not present at all). This extra flexibility allows the modelling of scenarios where participants dynamically join ongoing interactions — which are referred to as conversations. The work [4] extends [21] by introducing a notion of “behavioural role”, which has also been exploited for the purpose of security analysis, as we briefly report at the end of Section 4.

3. Security Types for Communication-Centred Calculi

3.1. Introduction

An increasingly relevant security issue is that of preserving the confidentiality of private data that are 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 to untrusted users of legally accessed data, thus ensuring end-to-end confidentiality. Hence, compared to access control, secure information flow may be viewed as additionally restricting the access to 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 3.2 and 3.3 respectively.

Another important security property is data integrity, which is often presented as the dual of confidentiality and may similarly be 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 originate from untrusted sources. Type systems for integrity are discussed in Section 3.4.

Two further aspects related to security are reputation and runtime adaptation:
• Security may be intended in the broad sense of “quality of service”. In that case, reputation systems, which guide principals in the choice of partners with whom to interact within communication protocols, may also help achieving security. A typed multiparty session calculus where participants’ “interaction histories” are used to compute their reputations, is overviewed in Section 3.5.

• As communication-centric systems may be subject to varying requirements and/or be deployed on top of dynamic platforms, runtime adaptation mechanisms are called for. These mechanisms should be disciplined so as to rule out dynamic updates that compromise 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. Section 3.6 describes a process framework which exploits behavioural types to jointly enforce runtime adaptation and the combination of access control and secure information flow.

3.2. Access Control

The work [61] presents a type system for COWS [69], a formalism for specifying and combining services, while modelling their dynamic behaviour. This type system allows 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, let us consider a standard buyer-seller-bank protocol, in which:
1. the buyer asks an item to the seller and receives a price from the seller together with a bank account number;
2. the buyer may either accept the price and send a credit card number to the bank, or turn down the offer.

In this scenario, it is clear that only the buyer and the bank, but not the seller, should have access to the credit card number. Therefore, the type system of [61] validates processes implementing the protocol described above, but not variants in which by mistake the buyer would send the credit card number to the seller.

The work [60] enriches the calculus of [16] (a variant of SCC [15]), with security levels for controlling access rights. In the original calculus, communications can either follow fixed protocols or use pipelines. In the new calculus, processes are framed by security levels. A process framed by a level $\ell$ can exercise rights of security level not exceeding $\ell$. 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 [60] comes equipped with a type system that statically ensures these security properties.

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

The treatment of access control in [23] is similar to that of [60], since both participants and data have security levels, but it enhances security by taking advantage
of the mechanism of delegation. In fact, the basic model is the calculus of multiparty sessions with delegation first introduced in [54] (in the variant defined in [11], where no linearity check is needed since channels are identified by session participants). In this calculus, instead of sending a bank connection to the buyer, the seller delegates to the bank the part of the interaction dealing with the credit card. The type system which assures access control has an explicit type constructor to track delegation, which allows the delegated part of a session type to be marked. A main feature of this calculus and type system is a form of data declassification. In the previous buyer-seller-bank example, the bank should be allowed to tell the seller whether the credit card number of the client is valid or not - which implies a declassification, of course without revealing the number itself. Declassification here is tailored to communication protocols, since data can be declassified only while they are being exchanged between two parties, at the condition that the receiving partner has the expected credential.

The comparison between the three access control approaches just described is not easy, since the underlying calculi offer different interaction patterns, namely a primitive for session killing in [61], a pipeline constructor in [60] and channel delegation in [23]. In our view, the most interesting problems for data protection arise with delegation, since it allows a transparent change of ownership for a given communication channel.

3.3. Secure Information Flow

As already hinted at above, the work in [23] considers a calculus for multiparty sessions with delegation, enriched with security levels for both participants and data, and defines a noninterference property for it, formalising the preservation of data confidentiality. It then proposes a session type system for the calculus, introducing secure information flow requirements in the typing rules in order to ensure simultaneously 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. This study revealed an interesting interplay between the constraints used in security types and those used in session types to ensure properties like communication fidelity and progress.

In [23], the notion of security is based on the observation of messages while they are being exchanged. The observation power depends on the level of the observer. An observer of level \( \ell \) can only see messages of security level lower than or equal to \( \ell \). For simplicity, we assume here just two security levels \( \top \) and \( \bot \) (although [23] 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”).

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

\[
s^+? (x^\top), s^+! < 1^+ >
\]
emits the low output “1” only if it first receives the high input from the environment. However, this kind of information leak can be sanitised by rendering inputs and outputs persistent, so that high messages are always offered when needed. Hence the first kind of security flaw, caused by different high inputs, seems more fundamental.

A subsequent paper [22] moves one step further by equipping the above calculus with a monitored semantics, which blocks the execution of processes as soon as they attempt to leak information. The safety property induced by this monitored semantics is shown to strictly imply the security property. Indeed, safety is a property of individual computations while security is a property of the set of computations of a process, which may hold even if some of these computations are unsafe.

The approach in [22] may be summarised as proposing three increasingly precise means for tracking information leaks in sessions: a syntactic property (typability), a local semantic property (safety) and a global semantic property (security).

We illustrate the differences between typability, safety and security by means of a simple example that should convey the intuition, although it does not completely fit the definitions of typability and safety in [22]. We convene here that typability requires the absence of any “level drop” from the expression tested by a conditional to a subsequent communication, while safety requires the same condition but only in computations that may actually occur. Instead, in [22] the “level drop” is forbidden only from inputs to subsequent communications (this is sufficient to imply the absence of level drops also in conditionals, because any conditional whose tested expression is high and open must be preceded by a high input), but an example along these lines would require recursive participants for the overall process to be secure, as argued above, and thus it would be too complicated for the present discussion.

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

$$\text{if } \text{true}^\top \text{ then } s^+! < 1\bot > \text{ else } s^+! < 2^\top >$$

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

$$\text{if } \text{true}^\top \text{ then } s^+! < 2^\top > \text{ else } s^+! < 1\bot >$$

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

**Discussion**

There appears to be an influence of classical session types upon security types. Indeed, one of the main causes of insecure information flow in a concurrency scenario is the possibility of different termination behaviours in the branches of a high conditional (a conditional which tests a secret expression). This may give rise to so-called termination leaks. In session calculi, there are three possible termination behaviours: proper termination, deadlock and divergence. Then, a termination leak may occur for instance
if one branch of a high conditional terminates while the other diverges or deadlocks, assuming successful termination is made explicit by an observable action. Session types help containing this phenomenon, by imposing some uniformity in the termination behaviours of conditional branches: for instance, a terminating branch cannot coexist with a diverging branch, 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. It is worth mentioning that the session type system studied in [19]—which is the basis for the dependent session types reviewed in Section 5—ensures process termination by typability, as shown in [64].

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

\[
\text{IF } x^T \text{ THEN } s^+! < 1^T > \text{ ELSE REC } X.s^+! < 2^T > .X
\]

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

\[
\text{IF } x^T \text{ THEN } s^+! < 1^T > \text{ ELSE } s^+! < x^T + 3 >
\]

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

\[
\text{IF } x^T \text{ THEN } s^+?(y).r^+?(z).s^+! < 1^T > .r^+ < 2^T > \\
\text{ELSE } s^+?(y).s^+! < 1^T > .r^+?(z).r^+ < 2^T > \\
| s^-! < 3^T > .s^-?(t).r^- < 4^T > .r^-?(u)
\]

Here, if the THEN branch is taken the process will terminate successfully, while if the ELSE branch is taken the interaction will deadlock. These global deadlocks are forbidden by the more refined behavioural type systems in [59, 61, 63].

### 3.4. Integrity of Communicated Data

We start considering the standard user-ATM-Bank example [53]. 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:

\[
\begin{align*}
\text{user} & = s^+! < \text{userId} > .s^+! < \text{depositAmount} > \\
\text{ATM} & = s^-?(\text{userId}).s^-?(\text{depositAmount}). \\
& \quad r^+! < \text{userId} > .r^+! < \text{depositAmount} - 10 > \\
\text{bank} & = r^-?(\text{userId}).r^-?(\text{depositAmount})
\end{align*}
\]

can be typed, since the first message sent by the user has type String and the second message has type Int, as expected by the other participants.
In order to cope with such kind of misbehaviour, correspondence assertions [48] are incorporated in the theory of session types [13]. Two correspondence assertions can be paired by the keywords BEGIN, END and have values which allow the integrity of the communicated data to be checked (in this example userID and depositAmount). The user and the bank processes with correspondence assertions become:

\[
\begin{align*}
\text{user} &= \text{BEGIN(userID, depositAmount).} s^+! < \text{userID} > .s^+! < \text{depositAmount} > \\
\text{bank} &= r^\sim?(\text{userID}).r^\sim?(\text{depositAmount}).\text{END(userID, depositAmount)}
\end{align*}
\]

thus allowing the cheating ATM to be discovered, since the operational semantics requires the same values in paired correspondence assertions.

Session types with correspondence assertions can be used to check:

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

More recently [5] presents a \(\pi\)-calculus with assume and assert operations, typed using a session discipline that incorporates refinement formulae written in a fragment of Multiplicative Linear Logic [46]. 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.

### 3.5. Reputation

The work [14] introduces a notion of reputation into session calculi, with the aim of regulating the participation of principals in sessions depending on the history of their past interactions. To this end, it uses the multirole session calculus of [30], where each role may be inhabited by a varying number of participants. The reputation associated with principals in a service is built on the basis of their behaviour as participants in past sessions of the service. The service checks the reputation of principals before allowing them to take part in a new session. Symmetrically, principals can declare their own policies, and check them against the reputation of the current participants before deciding whether or not to join the service.

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

\[
\begin{align*}
s^+! < \text{item} > .s^+? < \text{price} >. \text{IF good(price) THEN } s^+!\star < \text{ok} > \text{ ELSE } s^+!\star < \text{ko} >
\end{align*}
\]

where the sending actions decorated by \(\star\) are recorded in the buyer’s history. From the point of view of the seller, a buyer who has a long record of purchases will be more interesting than a buyer who has a long record of refusals. Then, for instance, the seller will be inclined to propose special offers to the first buyer but not to the second.

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

As (communication-centred) software systems rely on highly dynamic infrastructures, such as those based 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 runtime adaptation and security requirements: 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; on the other hand, one would like adaptation mechanisms which address functional requirements but also which 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 relation between security and runtime adaptation, a comprehensive approach that exploits their similarities and complementarities appears natural. This is the main motivation of the paper [23], which integrates security guarantees (access control and secure information flow) and self-adaptation within a process framework of multiparty structured communications. More precisely, behavioural types with security levels are used to monitor reading and writing violations, corresponding respectively to access control violation and information leaks. Behavioural types define security policies by stipulating reading and writing permissions, represented by security levels. While a reading permission is an upper bound for the level of incoming messages, a writing 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 reading or writing permission. An associated operational semantics is instrumented so as to trigger adaptation mechanisms in case of violations but also to prevent the violations to occur and to propagate their effect in the choreography.

The framework of [23] consists of a language for processes and networks, global types, and runtime monitors. Runtime 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.

We first explain the local adaptation mechanism. In case of a reading violation, the local adaptation mechanism modifies the behaviour of the monitor so as to omit the disallowed read, and then injects a process compliant with the new monitor. In case of a writing violation, the local adaptation mechanism penalises the sender by decreasing the reading level of his monitor and replacing the implementation for the receiver.

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 around 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, 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), then one may 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 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.

4. Security for Dynamic Web Data

4.1. Introduction

In an open distributed network, it is extremely important to provide security and protect privacy during transfer and management of data. These issues are reviewed for dynamic web documents handled by XML (Sections 4.2, 4.3) and web of data (Linked Data) published and linked using Resource Description Format (RDF) (Section 4.5).

For a given security policy of a distributed system containing semi-structured XML documents, the aim is to provide that the system behaves according to the prescribed security policy. In a calculus with suitable type system, security can be verified by typing, as it is presented in [32, 33, 44, 45].

Linked Data provides some sensible guidelines for publishing and consuming data on the Web. Data published on the Web has no inherent truth, yet its quality can often be assessed based on its provenance. Building on [55], the paper [35] provides a calculus of processes which use, consume and publish Linked Data tracing provenance.

4.2. A Calculus for Modelling Dynamic Web Data

The Xdπ-calculus is introduced in [41] as a formal model for reasoning about dynamic web data. A network of peers is modelled as a parallel composition of locations:

\[ N ::= \quad 0 \quad \mid \quad N \parallel N \quad \mid \quad I(T \parallel P) \quad \mid \quad (\nu c)N, \]
where each location consists of a data tree \((T)\) and a process \((P)\). Distinct locations can share communication channels.

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

\[
T ::= \emptyset_T \mid T \mid T \mid a[T] \mid a[\square P] \mid a[p@l]
\]

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

Processes are:

- the ones for modelling local communication from the \(\pi\)-calculus,
- the go \(l.P\) for movement of processes between locations (from d\(\pi\) [51]),
- the novel run\(_p\) that locally activates the static processes identified by the path \(p\),
- the novel update\(_p\)(\(\chi; V\)).\(P\) that locally updates the local data tree, where \(\chi\) is a pattern and \(V\) is a data term (we can also assume to have copy\(_p\)(\(\chi\)).\(P = \text{update}_p(\chi, \chi).P\) and paste\(_p(T)\).\(P = \text{update}_p(x, x|T)\).

Hence, there are three kinds of interaction: local interaction between processes, local interaction between processes and data trees, and interaction between locations. Given this focus on interaction, it seems that the line of work described here may be further explored using techniques from behavioural type systems.

### 4.3. Security of Web Data

In [33][54], the authors introduce a type system for the Xd\(\pi\) calculus, in order to control communication of values, access to data and migration of processes between locations. The type system is based on types for locations, data, and processes, expressing security levels. Security levels are taken from a partially ordered set \(\{i, j, h, \ldots\}, \leq\). Location names are decorated with security levels. A tree can store in its leaves data of different security levels and it is enclosed in a location of independent security level. The access to data and the mobility of a process depend on the security level of its source location. The source location is the one where the process was in the initial network or where the process was activated from a script.

In a well-typed Xd\(\pi\) network, the following security properties hold. For security levels \(i, j, \) and \(h\),

- a channel within a process whose source location has level \(h\) can communicate only values whose security levels are less than or equal to \(h\);
- a process whose source location has level \(h\)
  - can migrate to a location of level \(j\) only if \(j \leq h\),
  - can copy from the local tree only data of level \(j\) with \(j \leq h\), and
  - can modify in the local tree only data of level \(j\) with \(j < h\);
- a static process of level \(j\) which is contained in a leaf of a tree in a location of level \(i\) can be activated only if \(j \leq i\).
Consider a simple distributed network consisting of an online library, a store, a guest, a member and an owner. Let the library, written in Xd\π notation, be

\[
\text{library}^1 \parallel T_{\text{library}} \parallel \text{P}_{\text{library}}
\]

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

\[
\text{copy}_{\text{author/book/point}} (y@x^2). \text{go} x. \text{copy}_{\text{y}} (z^2). \text{go library}^1. \text{paste}_{\text{author/book/get}} (z^2)
\]

(with patterns decorated with security levels).

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

- the guest is allowed to go to the library and copy the title, as specified by \( \text{go library}^1. \text{copy}_{\text{author/title}} (\chi) \);
- the member is allowed to go to the library and get the book, by the process \( \text{go library}^1. \text{run}_{\text{author/book/get}} \);
- and the owner can go to the library and change its data by \( \text{go library}^1. \text{update}_{\text{author/book/get}} (\chi, \text{abstract}\{\emptyset T\}) \).

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

### 4.4. Role-based Access Control of Web Data

In [32], the Xd\π-calculus is equipped with role-based access control, with dynamic administration of permissions given to roles, and named \( \Box \text{Xd\π-calculus} \). A lattice of roles is assumed. Each location has a policy and consists of a process with associated roles and a data tree with roles assigned to edge labels, representing permissions (to access edge labels) assigned to roles. Pure (without roles) processes are Xd\π processes extended with commands (enable and disable) for administration of roles assigned to edge labels. A process is a parallel composition of pure processes with associated sets of roles. A role can be assigned to different edges and different processes, and the behaviour of the system is controlled by roles.

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

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

- A channel within a process can communicate only values with at least one (characteristic) role lower than or equal to one role assigned to the process.
• A process
  – can migrate to a location only if it complies with the policy of that location.
  – can read (copy) and change a data in the local tree only if the data is access-
    sible to the process.
  – can add (enable) or delete (disable) a role associated to an edge in the local
    tree only if this is allowed by the location policy.
• A script is activated in a location only if the corresponding process with roles
  respects the policy of that location.
• A tree built in a location by a change, enable or disable command respects the
  policy of that location.

For the set of roles \{\text{guest}, \text{member}, \text{owner}\} ordered as \text{guest} \subseteq \text{member} \subseteq \text{owner},
consider the online library:

\[
\text{library} \left[ \text{author}^{\text{guest}} [T_1 | T_2] \parallel R_{\text{library}} \right],
\]

where

\[
T_1 = \text{title}^{\text{guest}} [T],
T_2 = \text{book}^{\text{guest}}[\text{pointer}^{\text{member}} [T@\text{store}]|\text{download}^{\text{member}} [\square R]],
\]

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

\[
(\{\text{guest}\}, \{\{\text{owner}\}, \text{guest}\}, \{\{\text{owner}\}, \text{member}\})
\]

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

\[
R_{\text{library}} = \text{enable}_{\text{author/book}}^{\text{guest}}_{\text{owner}}
\]
gives the permission to access the edges \text{pointer} and \text{download} in the library to the role \text{guest}. The subtree \(T_2\) in \text{library} becomes:

\[
T_2 = \text{book}^{\text{guest}}[\text{pointer}^{\text{guest}} [T@\text{store}]|\text{download}^{\text{guest}} [\square R]].
\]

A notion of subtyping is given in \[58\] so as to add extra flexibility to the type system.

4.5. A Type System for Provenance Based Access Control

Linked Data recommends the data format to be based on triples of Uniform Re-
source Identifiers (URIs). The protocols for Linked Data allow triples to be retrieved
from locations and written to other locations. Thus a history of ‘where and who’ provenance can be accumulated. Each time an agent publishes data in a location, the agent and location can be recorded in the provenance history of the triple. Furthermore, the data may be processed locally, by the agent. Recording the operations that were applied to the data provides a notion of ‘how’ provenance. The paper [35] introduces a calculus that deals with provenance for Linked Data. For example, the following provenance trace represents that initially there were two pieces of data. One piece of data was published by agent ACM in [acm, Toyama], another piece was published by agent CiteSeer in [cs, Toyama]. Agent RKBExplorer consumes both pieces of data, applies the function Clean to the combination of both pieces of data, and publishes it in location Toyama.

\[
\text{RKBExplorer} \cdot \text{Clean} \cdot ((\text{ACM} \cdot \text{Toyama}) \lor (\text{CiteSeer} \cdot \text{cs})),
\]

Locations are equipped with policies prescribing which agents can read/modify data. A type system for the calculus is defined. The type system guarantees that access control policies for data are respected by processes run by agents. The access control policies are based on the provenance of the data. More precisely, the typing rules assure:

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

The issues addressed in this section 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 (e.g., Alice and Bob). More recently, roles have also gained a behavioural denotation, 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. In [45] a first step in characterising the delegation of authorizations to impersonate roles is taken, building on techniques presented here and on the behavioural type system given in [4].

5. Logical Approaches to Security based on Behavioural Types

5.1. Introduction

Session types consist of high-level specifications of the communication behaviour of distributed, concurrent processes along bidirectional 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., [13] mentioned previously and [73]). Recently, logical foundations for
session types have been established via Curry-Howard correspondences with linear logic \cite{19}. 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 us to express and enforce complex properties of data transmitted during sessions \cite{75}. 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 \cite{49}), we are able to specify arbitrary properties of the communicated data in such a way that the proof objects that witness the desired properties are themselves exchanged during communication. Moreover, the solid logical foundations of the approach allows for further (logically grounded) extensions to the data language to capture features of interest in an almost immediate way, such as digital proof certificates and proof object erasure through modal affirmation and proof irrelevance \cite{66}.

5.2. Linear Logic and Dependent Session Types

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

The work of \cite{19} interprets the propositional connectives of linear logic as the session types assigned to $\pi$-calculus channels in such a way that linear logic proofs can be interpreted as typing derivations for $\pi$-calculus processes. Moreover, the computational procedure of proof simplification or proof reduction is directly mapped to inter-process communication, thus obtaining a true correspondence between the dynamics of proofs and the dynamics of communicating processes. The connectives of linear logic are linear implication $A \multimap B$, which is interpreted as the input session type (i.e., input along $c$ a fresh session channel $d$ that will behave as $A$ and proceed along $c$ with the continuation type $B$); its dual, multiplicative conjunction $A \otimes B$, which is naturally interpreted as session output (output a session channel of type $A$ and continue as $B$); the multiplicative unit, $1$, denoting the inactive or terminated session; additive conjunction $A \& B$ denoting an offer of a choice, meaning that a session of type $A \& B$ will be able to offer along the session channel either $A$ or $B$, the choice of which is left to the session client; dually, additive disjunction $A \oplus B$ denotes alternative behaviour, and so a session of type $A \oplus B$ will unilaterally choose to behave as either $A$ or $B$. Finally, the linear logic exponential $!A$ is mapped to replication, in which a session of type $!A$ will offer a potentially unbounded number of instances of the behaviour $A$. 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.

Recently, \cite{75} extended this framework of propositional linear logic as session types to incorporate dependent session types by moving to a first-order setting, introducing the two quantifiers $\forall x : \tau. A$ and $\exists x : \tau. A$, where $x$ may occur free in $A$. The quantification variable is itself typed, with a domain of quantification $\tau$. The language of terms inhabiting $\tau$ is a typed $\lambda$-calculus, which is left as general as possible, with
the usual soundness requirements of progress, substitution and type preservation. The
interpretation of these session types is (typed) term output for the existential \(\exists x:\tau.A\)
and term input for the universal \(\forall x:\tau.A\). Thus, a session of type \(\exists x:\tau.A\) outputs a term
\(M\) of type \(\tau\) and proceeds as type \(A\{M/x\}\), whilst a session of type \(\forall x:\tau.A\) behaves
in a dual manner.

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

\[
\text{UpInterfaceP}(x) \triangleq x : \forall n:\text{int}.\forall p:n > 0.\exists y:\text{int}.\exists q:y > 0.1
\]

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

5.3. Proof Irrelevance

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

To model the possibility of omitting proofs at runtime, the work of \cite{75,66} extends
the framework by internalising into the proof object language the concept of proof
irrelevance \cite{65}, through a modality which we write \([\tau]\), which types terms of type \(\tau\)
that can be safely erased at runtime. This notion of erasure safety essentially means
that such terms can never be used to compute values that are not themselves erasable.
For instance, the type above can be rewritten as:

\[
\text{UpInterfaceI}(x) \triangleq x : \forall n:\text{int}.\forall p:n > 0.\exists y:\text{int}.\exists q:y > 0.1
\]

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

\[
\forall x:\text{unit}.A \cong A
\]
\[
\exists x:\text{unit}.A \cong A
\]

to erase the communication overhead in a safe and logically sound way. An alternative
technique familiar from type theories is to replace sequences of data communications
by a single communication of pairs. When proof objects are involved, these become \(\Sigma\)-types which are inhabited by pairs. For example, we can rewrite \(\text{UInterface}_2\) as:

\[
\text{UInterface}_2(x) \triangleq x : \forall p : (\Sigma n : \text{int}.|n > 0|).\exists q : (\Sigma y : \text{int}.|y > 0|).
\]

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

### 5.4. Affirmation and Digital Certificates

The examples above showcase what can be seen as two extremes in a spectrum of trust. In the original example, no trust between the parties is assumed and therefore all proof objects must be made explicit in communication at runtime. On the other hand, proof irrelevance models a scenario of full trust, where no proof objects are expected at runtime. In practice, there are trade-offs between trust and fully explicit proofs. For instance, when downloading a large application we may be willing to trust its safety if it is digitally signed by a reputable third party, but if we are downloading and running a piece of Javascript code embedded in a web page, we may insist on an explicit proof that it adheres to our security policy. To make these tradeoffs explicit in session types, [66] 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 \(\Diamond_K \tau\) to the framework, which types objects that asserting the property \(\tau\), signed by principal \(K\) using its private key. An affirmation object is built by taking the original proof object that asserts \(\tau\) 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 \(\tau\) as true.

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

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

which expresses the type of a service that inputs a function \(f\), accepts a verifier’s word that it is decreasing (denoted by the object \(p\), which is a certificate of that fact) and returns a fixed point of \(f\) to its client. In realistic scenarios such as proof-carrying file systems [42], this approach of using affirmation and proof irrelevance results in substantial less overhead in communication when compared to proof-carrying code in the sense of Necula and Lee [62], where the proof objects become too big to be transmitted and checked every time a file is accessed.
5.5. Dynamic Spatial Logics

Reasoning about security often requires describing the structure of systems so as to express the desired properties. For example, one may say that a secret is a piece of information known by a part of the system and unforgeable by other parts [17]. Specification logics that allow us to talk about the structure of systems, such as dynamic spatial logics (see [17] for a survey), may then be particularly suited to express security properties in a natural way. Furthermore, combining the ability to inspect the structure of systems together with the ability to talk about system behaviour, one may then reason about security properties in the context of dynamic concurrent systems.

The relation between dynamic spatial logics and behavioural types for security is then twofold. On the one hand, the dynamic properties that characterise system behaviour are necessarily related to the behavioural characterisations carried by the types. On the other hand, the structural or spatial characterisations may help describe security properties in a natural way, namely by allowing to talk about separate parts of the system.

In the last years a number of type-based verification approaches based on dynamic spatial logics have been proposed. Focusing on concurrency and resource control, the spatial behavioural types presented in [18] allow for the description of resource dependencies and resource ownership in a distributed object model. Focusing on communication safety properties, [2] combines ideas from dynamic spatial logics and the generic type system presented in [57] to analyse properties such as (communication) race freedom, responsiveness and deadlock freedom. The approach presented in [2] relies on model-checking performed at the type level, where types provide a spatial-behavioural abstraction of systems, shown decidable for an interesting class of properties in a subsequent work [3]. Also related to dynamic spatial logics is the type discipline of behavioural separation introduced in [20] for controlling interference in concurrent imperative programming, where types mix both behavioural/temporal and structural/spatial characterisations.

Although these type systems do not focus on security properties, they seem to provide an interesting basis to build on so as to reason about security properties in dynamic concurrent systems. In fact these types characterise behaviour and spatial distribution which seems like a natural setting to address security properties, that talk about what are the allowed behaviours in the several parts of the system.

6. Secure Interactions with Untrusted Components

6.1. Introduction

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

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

### 6.2. A Secure Protocol Compiler

Based on a restricted session typed language, the works [27][28] offered a first answer to these questions. The proposed language, expressed as a local type language with a global graph-like representation, does not support any asynchrony or parallelism — indeed this typed language is sometimes referred to as **sequential multiparty session types**.

The principle of [27][28] is to use the session specification to generate a cryptographic protocol (and its implementation) that will protect the trusted participants against any coalition of compromised peers. The idea is that, in order to ensure that an incoming message is valid with respect to the session specification, that message should carry enough trustworthy information to be able to prove that the protocol history was compliant up to that point. Technically, this is achieved by means of asymmetric cryptography, using signatures of past messages to convince the receiver that the specification was followed by all participants. The minimal (necessary and sufficient) set of signatures to be transmitted and checked are defined in [27] through the notion of the visibility. The protocol also relies on other cryptographic primitives, such as nonces and a cache system to prevent replay attacks between session instances or within a given session.

The formal security notion proved in [27][28] 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.

The work [26] presents a prototype implementation of the above approach as an extension of OCaml. A compiler which takes as input a session description and produces an OCaml module with a function for each participant is developed. Any user code calling one of these functions is guaranteed through the standard ML type system to statically follow the appropriate local session type. This is achieved through a monadic programming style. The module’s cryptographic implementation then guarantees that, even in the case of compromised peers, all the messages seen by uncompromised participants are consistent with the session specification.

The work [67] extends [28] by considering a more expressive language featuring concurrency and synchronisation within session runs. Their specification language, however, drifts from a session typed language to a CCS-like protocol language whose implementation is represented as a set of traces.

In [12], a different approach is taken to improve the work in [26] with a simpler and more efficient cryptography (using a combination of asymmetric and symmetric
cryptography), and extend the session description language with value annotations. This extension allows one to model commitments and to protect independently the integrity of each payload. As in [26], a compiler implementation is realised, which relies on OCaml typing for local protocol conformance, and on a generated optimised cryptographic protocol implementation for session integrity.

### 6.3. Contract-oriented Service Composition

In the top-down approach to the design of session-typed distributed applications [54], a choreography describing the global interaction behaviour of the application is projected to a set of local types, which describe the roles of each single participant in the application. Each participant provides an implementation of its role: if all these implementations respect their local types, then 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 requiring to inspect the code of each participant is then pointless: consequently, the 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 [56]. Once these sessions are created, participants can perform the actions prescribed by their contracts (in case they are session types, this would result in doing 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 culpable at each step, i.e., responsible for the next interaction. Systems developed under this design approach are called contract-oriented systems [59].

While untrusted participants may cause deadlocks in the contract-oriented approach as well as they do in the top-down one, a difference between the two approaches is that, in the first case, one can use contracts to single out the participants which have breached the agreement, so causing the deadlock. This can be associated with sanctions inflicted to the culpable participants; these sanctions could range e.g., from lowering the participants’ reputation, to removing them from the repository of available services, or imposing them a fine.

### 6.4. Honesty in Contract-oriented Systems

Interacting with ill-typed or untrusted participants may have non-obvious consequences, especially in case of applications with multiple interleaved sessions. For instance, consider a participant A dealing with sessions s and t, involving respectively participants B and C: if s gets stuck because B misbehaves, then A may become unable
to advance on $t$; because of this, $C$ may become stuck and unable to advance on another session — and the problem may cascade to other sessions and participants. This is especially problematic if interactions have some economic relevance, and thus $A$’s failure to progress in $t$ may damage $A$ itself, or $C$, or others. Therefore, $B$’s misbehaviour on $s$, albeit not necessarily malicious, can be regarded as an attack; it is thus reasonable that $A$ may want to limit the disruption that $B$ might cause.

A desirable goal for the designer of $A$ would be to guarantee that $A$ is never responsible for some session being stuck — and therefore, even if $s$ is blocked by $B$, $A$ will still behave according to her role in $t$. This property — called honesty — is formally defined and investigated in the contract-oriented specification language $CO_2$ [10]. Honesty can be seen as multi-session well-typedness: if a participant is honest, his process will behave according to his contracts in each session he will establish, even if other participants will not cooperate.

In general, the goal of a developer would be that of publishing only honest services, which always respect contracts — also 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), analysis techniques are in order to detect honesty of processes. While honesty is not decidable in general [10], 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 type systems [8], and model checking [7].

6.5. Protection Against Untrusted Brokers

In contract-oriented applications, participants advertise contracts to some broker, which establishes session among participants whose contracts admit an agreement. In such 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 her objectives, then some other (dishonest) participant will be culpable of a contract infringement.

In the above workflow, it is crucial 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 this context, contracts should allow participants to reach their goals in contexts where the other participants are cooperative, and prevent them from performing imprudent actions which could be exploited by malicious participants, in untrusted contexts.

This problem has been addressed in [6] in a game-theoretic setting, where session interactions are interpreted as games over event structures (ES [80]). In this setting, a participant wins in a play (a trace of the ES) when she reaches success, or some other participants can be blamed for a violation. The intuition is that the infrastructure will eventually inflict sanctions to the participants who have violated their contracts.
Two key notions in this model are that of agreement and protection. Agreement is a property of contracts which guarantees that each honest participant may reach her objectives whenever the other participants cooperate. Moreover, if an honest participant does not reach her goals, then some other participant can be blamed. A contract protects its participant if it guarantees at least a non-losing strategy whenever the contract is composed with any other contract (possibly that of an adversary).

The notion of agreement in the game-based model is related with the notion of compliance in (binary) session types: more precisely, two session types are compliant if and only if, in their encoding as event structures, the eager strategy (which prescribes a participant to do all her enabled events) is winning for both participants. Hence, compliance implies agreement, while the converse does not hold: there can still exist a winning non-eager strategy. For instance, for $P = \text{pay!} \otimes \text{receive!} \oplus \text{abort!}$ and $Q = \text{pay?} + \text{receive?}$, $P$ and $Q$ are not compliant, but $P$ agrees since choosing the branch $b!$ leads $P$ to win.

If brokers are dishonest, then [1] shows it is not always possible to obtain both that agreement and protection. For instance, assume a participant $A$ with session type $P = \text{pay!} . \text{receive?} \oplus \text{abort!}$. If the broker makes $A$ interact with another participant with session type $Q = \text{pay?}$ (which is not compliant with $P$), then to be protected $A$ should avoid firing $\text{pay!}$, because doing so will never lead to the expected $\text{receive?}$. In this example, a strategy which protects $A$ would be the one which only enables $\text{abort!}$. Reconciling agreement with protection is still possible by relaxing the classical notion of causality, and assuming that some events can be done in the absence of a causal justification in the past, provided they have one in the future [6].

7. Gradual Security Types and Sessions

7.1. Introduction

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

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

7.2. Gradual Typing and Security

One approach to address these problems is to resort to gradual typing [71] [70], which has received a lot of attention. Gradual type systems have been developed from dynamic type systems [1] [50]. In a language with a dynamic type system, values of
compatible types may be coerced. Specifically, a value of arbitrary type $T$ may be injected in a special type \texttt{Dynamic}. This injection constructs a pair of a run-time representation of $T$ and a value of type $T$. There are also projections from type \texttt{Dynamic} to any type $T$, but these projections may fail if the underlying value is not paired with the representation of $T$. Such a system may be embedded in a conventionally typed language \cite{[1]} or coercions may be used to specify and optimise a dynamically typed language \cite{[50]}.

In the extreme case, the language is dynamically typed (like Scheme or JavaScript) and each arithmetic operation, say, has to coerce each argument to a number before it can execute. Among other approaches, coercion calculi have been used to address the inefficiency arising due to type tests where the outcome is known in advance \cite{[50]}. Gradual typing also addresses these efficiency issues. It finds a static typing for large parts of a program and inserts coercions in places where the static type check does not succeed. Gradual typing guarantees traditional type safety up to the violations detected by the inserted coercions.

The work \cite{[36]} shows that gradual typing is also applicable in a security context, albeit in the setting of the pure lambda calculus. Their approach has later been extended to an ML core language. This extension employs a very liberal treatment of references that are shared between statically and dynamically typed fragments \cite{[39]}.

Security type systems that control information flow and that track data integrity usually assume an underlying program that is well-typed according to some standard type system \cite{[79]}. 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.

Departing from gradual typing, gradual security typing assumes an underlying typed program. The gradual aspect of the system is restricted to the security labels in the existing work \cite{[36],[39]}. Gradual security typing guarantees termination insensitive noninterference \cite{[47]}. The statically security-typed parts have this property by means of the type system, whereas the dynamically security-typed parts have the property by means of a monitor that checks the no-sensitive-upgrade policy on run-time security labels.

In the best case, a coercion from static to dynamic adds run-time labels whereas a coercion from dynamic to static removes them. While such a design is possible in the presence of references, it restricts the use of references that are aliased between statically and dynamically typed parts of a program. For that reason, the language proposed in \cite{[39]} requires some dynamic checks even in the statically typed parts of a program.

Subsequent work on gradual annotated types \cite{[40]} 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 add run-time labels to all values, and monitor its execution. In contrast, new code would run without labels at full speed.
because its security properties are guaranteed by the static type system. Addressing the connection of legacy code with new code via communication channels is the place where session types or other behavioural types enter the scene.

7.3. Gradual Security Typing and Sessions

The work [81] shows that typestate can be subject to gradual typing in the concurrent object-oriented language Plaid [72]. The coercion into the dynamic type reifies the current typestate in a run-time value and runs an automaton that is synchronised with the transitions of the static typestate computation, similar to communicating automata [29]. They suggest to use the dynamic type during program development because their static typestate system requires program annotations to manage aliasing.

However, 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. However, it has been shown that linearity and gradual typing are largely orthogonal and that many results from standard gradual typing, in particular the blame theorem, carry over to a setting with linear types [38]. When restricting to affine typing, then the gradual aspects of the type systems may even be allowed to hide the affine property [76].

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

Thus, we are now in a position to actually build a system with gradual session types that also supports verifying security properties. It is expected that any of the existing static session systems with security awareness (e.g., [24] [22] [23]) can form the basis of a gradual system, similar as in the setting without sessions. First steps towards integrating gradual typing with session types have been taken [74]. Future work will take care of addressing linearity and gradual security.

8. 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 paper presents an overview of efforts to achieve this integration in a rigorous way, building upon core programming languages 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, data integrity, reputation, and runtime adaptation. We also discuss the security of web data based on XML documents, where the controlled mobility of both data and processes across distributed locations as well as forms of role-based access control and provenance are key issues. A fruitful research strand concerns logic-based approaches to behavioural types. Linear and spatial logics 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. Although combinations of gradual and session types are far from immediate, we briefly describe how initial steps towards a useful integration have been recently taken.

Acknowledgments We acknowledge support from COST Action IC1201 Behavioural Types for Reliable Large-Scale Software Systems (BETTY) and we thank BETTY working group 2 (Security) members for interesting related discussions.

References


