Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Towards a Safe and Secure web semantic framework

Supervisor:
Prof. Gianluigi Zavattaro

Presented by:
Carlo Prato

Co-supervisors:
Ilaria Castellani
Tamara Rezk

Session III
Academic Year 2018/2019

Sommario

Questa tesi descrive il lavoro da me svolto durante il mio tirocinio presso il centro di ricerca INRIA di Sophia-Antipolis all'interno del team INDES, sotto la supervisione di Ilaria Castellani e Tamara Rezk. Il team INDES si prefigge come obiettivi di ricerca lo studio di modelli e lo sviluppo di linguaggi per il **Diffuse computing**, un paradigma di computazione in cui è necessario gestire e mantenere strutture di calcolo distribuite su più nodi generalmente eterogenei e che possono non fidarsi l'uno dell'altro. In particolare INDES si concentra sullo studio dei diversi modelli di gestione della concorrenza che stanno alla base di questi sistemi e rivolge particolare attenzione verso il Multitier programming, un paradigma di programmazione emergente che mira alla riduzione della complessità nello sviluppo di applicazioni web, con l'adozione di un unico linguaggio per programmare le varie entità in gioco (client, server ed eventuali interazioni con un database). Molto rilevante nell'attività del team è il ruolo giocato dalle questioni di sicurezza (e particolarmente dalla necessità di proteggere la confidenzialità e l'integrità dei dati), la cui importanza cresce di pari passo con la diffusione di nuove tecnologie sempre più pervasive e comuni nella vita quotidiana.
Il mio tirocinio si è svolto nel contesto del progetto ANR CISC (Certification of IoT Secure Compilation) il cui obiettivo è quello di fornire, mediante l'utilizzo di tecniche formali, un insieme di semantiche, linguaggi e modelli di attacco per l'Internet of Things (IoT), termine che si riferisce ad un particolare tipo di sis-

temi composti da un insieme di dispositivi interconnessi, i quali interagiscono con l'ambiente in cui sono posti mediante diversi sensori ed attuatori. Durante la mia permanenza ho avuto modo di familiarizzarmi con diversi argomenti collegati alle attività del team, sia individualmente che durante seminari di diverso genere, come l'Information Flow, il Multitier Programming, aspetti legati alla Sicurezza di applicazioni web come la Web Session Integrity e le varie questioni di sicurezza specifiche all'IoT. Inoltre ho avuto l'opportunità di approfondire la mia conoscenza riguardo ai Session Types, una teoria di tipi volta all'analisi di correttezza di applicazioni distribuite. La mia ricerca individuale si è svolta nell'ambito di Webi, un framework semantico che mira ad una simulazione primitiva delle interazioni che avvengono tra server e client nel web, sviluppato da Tamara Rezk e da suoi colleghi. In particolare mi sono concentrato su un'estensione di Webi chiamata WebiLog, che permette di rappresentare sessioni autenticate e di formalizzare attacchi mirati a comprometterne l'integrità. Questo lavoro ha permesso di individuare diversi progetti futuri di ricerca, tra cui la definizione di Session Types per Webilog, il cui scopo è di garantire un'esecuzione corretta delle interazioni tra client e server, sia durante la fase di autenticazione che durante una sessione autenticata.

Summary

This thesis describes the work I did during my internship at the INRIA research center in Sophia-Antipolis, within the INDES team and under the supervision of Ilaria Castellani and Tamara Rezk.The main objectives of the INDES team is to study models and develop languages for **Diffuse computing**, a computing paradigm in which it is necessary to manage and maintain computing structures distributed on several heterogeneous nodes that usually do not trust each other. In particular, INDES focuses on the study of the different concurrency models that underlie these systems and pays particular attention to Multitier programming, an emerging programming paradigm that aims to reduce complexity in the development of web applications by adopting a single language to program all their components. The role played by security issues (and particularly the protection of confidentiality and integrity of data) is crucial in these applications, and ensuring security of web applications is another important goal of the INDES team. My internship took place in the context of the ANR CISC (Certification of IoT Secure Compilation) project, whose objective is to provide semantics, languages and attack models for the Internet of Things (IoT), a term that refers to systems composed of a set of interconnected devices, which interact with the environment in which they are placed by means of different sensors and actuators. During my internship I got to know several topics investigated in the team, both through my study of the literature and during seminars of different kinds: Information Flow,

Multitier Programming, aspects related to the security of web applications such as Web Session Integrity and various IoT specific security issues. I also had the opportunity to deepen my knowledge of Session Types, a theory of types aimed at analysing the correctness of distributed applications. My individual research took place within Webi, a semantic framework that aims at a primitive simulation of the interactions that take place between servers and clients on the web, developed by Tamara Rezk and her colleagues. In particular, I concentrated on an extension of Webi called WebiLog, which allows one to represent authenticated sessions and to formalize attacks aimed at compromising their integrity. This work allowed for the detection of future research projects, through which the definition of Session Types for WebiLog, whose purpose is to guarantee a correct execution of the interactions between client and server, both during the authentication phase and during an authenticated session.

Contents

Chapter 1

Introduction

Today's web applications combine several features. They rely on broad network accessibility, offer customizable digital environment while providing access to vast sources of information. These applications are sometimes called "diffuse" because of the multiplicity of interconnected computing devices, often mobile or portable, which constitute their computing environment. They require a combination of different programming paradigms, ranging from sequential computing up to parallel and concurrent computing. Since these applications rely heavily on sharing of private informations over networks, where mutually distrustful nodes are connected by possibly unreliable means of communication, there is a growing need for strong security guarantees.

The INDES team goal is to study models for diffuse computing and to develop languages for secure diffuse applications. The team aims to contribute to the whole chain of research for diffuse computing, including the study of foundational models, formal semantics as well as the design and implementation of new languages. Regarding the latter, emphasis is put on multitier programming, a programming approach aiming to ease the complexity of developing web applications. To this end, interest is placed towards the study of concurrency manage-

ment, since a deep understanding of different concurrency principles and models is required in order to be able to interface devices through a programming language. In the long term, the research conducted in the INDES team aims at providing scalable and sound language based techniques to be integrated into compilers for these languages, in order to enforce correctness and security guarantees within the execution of diffuse programs.

My internship was set in the context of the ANR project CISC (Certification of IoT Secure Compilation), whose goal is to define languages, semantics, attacker models, and security policies for the Internet of Things (IoT). The term IoT refers to a system of connected computing devices interacting in a number of ways with their environment. An example of an IoT system is "Smart Homes" technologies, where a multitude of devices is able to detect, thanks to embedded sensors, the occurrence of physical events, such as overheating, fires and many more, and react on the environment through actuators. There are a number of serious concerns related to security and safety aspects of IoT, especially due to the ability of these devices to remotely control physical elements of houses, opening the door to a number of serious threats from malicious network users. Several of these issues were discussed during the weekly seminars held by team members, named "reading groups", which included practical demonstrations of security attacks against smart watches, vacuum cleaners and web applications. This allowed me to dive into topics related to security of programs, including notions of Secure Information Flow and Web Session Integrity.

During my stay at INRIA I also had the chance to consolidate my knowledge and understanding of Session Types, a type theory for the analysis and verification of communication protocols in distributed systems. I also gave an introduction talk on this subject, within the regular seminar of the INDES team. Moreover, I participated in the BehAPI Summer School (Behavioural Approaches for API-

Economy with Applications), an event featuring theoretical and practical sessions on the concept of behavioural APIs, including several courses on Session Types held by experts on the topic. The school offered a mix of courses and boot-camps from academia and industry, supported by practical "hands-on" sessions with state-of-the-art tools and technology.

My personal research was carried out in the setting of Webi, a semantic framework which primitively simulates interactions between clients and servers on the web, developed by Tamara Rezk and her colleagues. One of my tasks was to enrich Webi in such a way that it could represent Authenticated Sessions as described in [18]. A further goal was to investigate the use of Session Types in this new instance of Webi to ensure safety and security properties for web sessions.

The rest of this document is structured as follows:

- In Chapter 2 the key concepts of Information Flow theory are introduced, such as the Lattice Model of security levels and the notion of noninterference, together with examples of the usage of these techniques in modern programming languages.
- Chapter 3 contains an in-depth analysis of the topics of Web Security that were covered during the internship, including a formalization of Web Session Integrity, possible attacks breaking this safety property and a discussion on IoT security issues.
- Chapter 4 deals with the topic of Session Types, explaining by examples their purpose and the properties they are able to guarantee.
- Chapter 5 explains the motivations and key points of multitier programming, presenting two languages that exemplify it: Hop and Links.

- Chapter 6 presents the Webi semantic framework as proposed by Tamara Rezk and colleagues.
- Chapter 7 defines WebiLog, an extension of Webi that accounts for authenticated sessions.
- Chapter 8 presents future research topics related to WebiLog. We scribble the basis of a type system based on Session Types. It also introduces a notion of noninterference and uses it to formalize the Session Integrity property.

Chapter 2

Secure Information Flow

The environment in which most web technologies are set nowadays is inherently heterogeneous and heavily relies on communication between various types of programmable and connectable devices, of which the proliferation and diffusion is now evident. Therefore, Web programming has become more complex and many issues have risen: device heterogeneity, concurrency, communication, mobility, untrusted code and the need for protection against several security attacks. Since computations occur concurrently between interconnected devices and it is possible that some of them are controlled by parties we may not trust, it becomes of crucial importance to define and enforce security policies on the data that are exchanged between them during communications.

We can target two fundamental security properties: **confidentiality**, which asserts that no divulgation of sensitive data happens during execution, and **integrity**, which asserts that no corruption of sensitive data is possible during execution.The theory of Secure information flow formalised through the notion of noninterference, covers relevant aspects of both these properties through static and dynamic techniques.

Ensuring secure information flow within programs in the context of multiple sen-

sitivity levels has been widely studied during the years, with seminal works from Denning [11], which introduced the Lattice Model and the associated static analysis technique, Goguen and Meseguer [13], who proposed the notion of noninterference, and Volpano, Irvine and Smith [25] who where the first to present a type system ensuring noninterference. In the following sections we will give a broad description of the Lattice Model and focus on the notion noninterference, which guarantees these properties.

2.1 The Lattice Model

In Denning's model an information policy is defined by a lattice $(SL, \leq)$, where SL is a finite set of security levels partially ordered by $\leq$. Security levels may be interpreted as confidentiality levels or as integrity levels. In the former case the bottom element is denoted by L (standing for low or public) and the top element is denoted by H (standind for high or secret). In the latter case the bottom element is denoted by U (standing for untrusted) and the top element is denoted by T (standing for trusted). Therefore, for any level l in one of these lattices we have $L \leq l \leq H$ and $U \leq l \leq T$. In the following we will mostly focus on confidentiality, but the same concepts are easily stateable for integrity. Security levels are assigned to objects containing information, such as variables. The security level of variable x is denoted by $\underline{x}$. This level is usually determined statically and not subject to update. A transfer of information from a variable x to a variable y is a permissible flow if $\underline{x} \leq \underline{y}$.

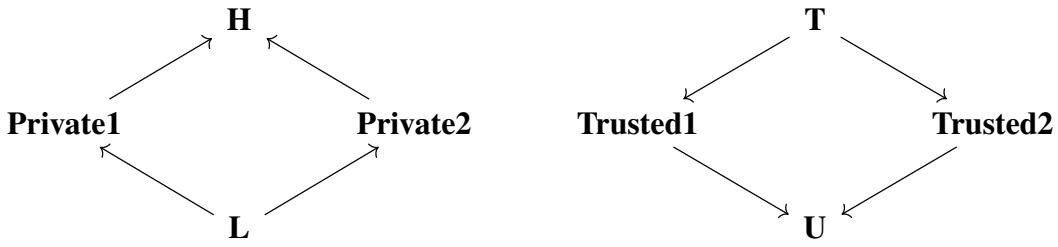

Figure 2.1: Security lattice for Confidentiality and Integrity.

To ensure Confidentiality only flows towards equal or higher levels are authorized as we can see on the lattice pictured on the left, while for Integrity of data, only flows of information towards equal or lower levels are authorized. All other flows are considered as insecure. In the case of confidentiality, insecure flows are also called information leaks. We may identify two different types of information flows, explicit and implicit. Explicit flows occurs as the result of executing any statement that directly transfers information to a location, such as an assignment or an I/O operation. Implicit flows occur as the result of execution or non-execution of a statement that causes an explicit flow to a location when conditioned on the result of an expression. Since these flows arise in different ways in a programming language, every construct needs a certification condition relating the security levels of its variables and data.

For example, the statement :

$$x := y$$

has the condition $\underline{x} \leq \underline{y}$ controlling an explicit flow. Conditions related to constructs like the **if** or **while** loops, control implicit flows like the one generated from the tested expression of a conditional to its branches. For example, the statement:

$$\mathsf{if}\ x > y\ \mathsf{then}\ z := w\ \mathsf{else}\ i = i + 1$$

contains an implicit flow from x and y to z and i. The statement has the condition

that $\underline{x} \oplus \underline{y} \leq \underline{z} \otimes \underline{i}$ where $\oplus$ and $\otimes$ denote respectively the least upper bound and the greatest lower bound operator.

2.2 Noninterference

Noninterference [13] is a security policy designed to ensure that objects and subjects at different security levels do not interfere with those at other levels. This policy enforces that an attacker should not be able to distinguish two computations from their outputs if they only vary in their secret inputs.

Noninterference aims at a strict separation of different security levels to ensure that higher-level variables values do not determine any lower-level variables values, in order to ascertain that the latter cannot be used to determine the former.

This leads to a very strict security regime and as a consequence, it is very difficult to create a program that completely meets all the demands of noninterference. An example is a password checker program that, in order to be useful, needs to disclose some secret information: whether the input password is correct or not, where the information that an attacker learns in case the program rejects the password is that the attempted password is not the valid one. Usually, this situation is referred to as the "No classified information at startup" exception.

We consider an l-observer as an entity which is able to observe variables of level l or lower. The objective of noninterference is to ensure that, by observing l'-level outputs for $l' \leq l$, an l-observer cannot reconstruct any l''-level input for $l'' \not\leq l$. This can be achieved by not allowing dependencies between l''-level inputs and l'-level outputs. To formalise this property, we introduce some notation. We assume states (or memories) s, s' to be functions from variables to values. We denote by $\langle c, s\rangle \downarrow s'$ the fact that command c running in state s produces state s'. Two states s_1, s_2 of a program are said to be l-equal, that is $s_1 =_l s_2$ if, for all

variables x with $\underline{x} \leq l$, $s_1(x) = s_2(x)$. We can now define noninterference for sequential programs as follows. A command c is l-noninterferent if, for all states s_1, s_2 such that $s_1 =_l s_2$, the following condition holds:

$$(\langle c, s_1\rangle \downarrow s_1' \wedge \langle c, s_2\rangle \downarrow s_2') \rightarrow s_1' =_l s_2'$$

This condition states that the execution of command c in l-equal states leads to l-equal states. A command is noninterferent, or secure, if it is l-noninterferent for any level l. There are two classes of techniques to ensure that security properties such as noninterference are satisfied by programs: dynamic techniques such as runtime monitoring, and static techniques which usually make use of a security type system to analyse source code. The latter approach was pioneered by Volpano, Irvine and Smith in [25] where a security type system was proposed and proven to be sound with respect to a noninterference property. We now comment some of the rules related to one of the two possible presentations of this type system, namely the one including subtyping. The type system assigns types (security levels) to both expressions and commands. The intuition behind this procedure is to establish limits within which a program can be considered as safe. Assigning a security level to an expression gives us an upper bound on the security levels of the variables that occur in it. On the other hand, assigning a security level to a command gives us a lower bound on the security levels of variables assigned to in it. The definition of a subtype relation allows for a secure arrangement of upwards information flows. We suppose Γ is an environment mapping variables to security levels $\tau, \tau' \in \mathrm{T}$ for a given security lattice $(\mathrm{T}, \leq)$.

$$\textsc{Assign}\ \frac{\Gamma \vdash e : \tau \qquad \Gamma(x) = \tau}{\Gamma \vdash x := e : \tau} \qquad \textsc{If}\ \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash c : \tau \qquad \Gamma \vdash c' : \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ c\ \ \mathsf{else}\ c' : \tau}$$

$$\textsc{Sub-Exp}\ \frac{\Gamma \vdash e : \tau \qquad \tau \leq \tau'}{\Gamma \vdash e : \tau'} \qquad \textsc{Sub-Com}\ \frac{\Gamma \vdash c : \tau \qquad \tau' \leq \tau}{\Gamma \vdash c : \tau'}$$

Rule ASSIGN ensures that the explicit flow from e to x is secure by requiring that they agree on their security levels, since τ appears on both hypotheses of the rule. An upward flow from e to x is still allowed by the system through subtyping: if $\Gamma \vdash e : L$ and $\Gamma(x) = H$, the type of e can be upgraded to H by Rule SUB-EXP and Rule ASSIGN can be applied with hypothesis $\Gamma \vdash e : H$. It is worth noting that the whole statement is given type H in order to be used in the context of other constructs, for instance in the branch of a conditional, which is secured by Rule IF. The intuition behind this rule is that c and c' are executed in contexts where information of level τ is implicitly known and thus they can only, for example, assign to variables of level τ or higher. Even if the rule requires the guard and both branches to have the same security level, it does not prevent an implicit upward flow from e to c and c', since subtyping can establish agreement by coercing the type of e to a higher level or the type of the branches to lower levels. No agreement can be reached if there is any downward flow from e, since the rule must reject the statement if this situation occurs. This model gives a precise operational characterization of the flow analysis: altering the initial value of a variable x with security level τ cannot affect the final values of any variable with security level τ' as long as $\tau \not\leq \tau'$.

2.3 Covert Channels in Programming Languages

Besides implicit and explicit flows which are detectable in programming constructs, many other features of a language could lead to leaks of information in the execution of a program. Such vulnerabilities are usually called side channels and are based on the principle that physical effects caused by the operation of a system can provide useful extra information about secrets inside the system.

A common side channel is represented by timing attacks which, observing how long some operations take to be performed, have been shown to be able to break cryptosystems. An effective countermeasure against such attacks is to design the software to be isochronous, that is to run in a constant amount of time, independently of secret values. This makes timing attacks impossible, but such countermeasures can be difficult to implement in practice, since even individual instructions can have variable timing on some CPUs. An example of a timing attack related to features in programming languages is presented in [23], where the authors take into account an important aspect of program runtime that is **automatic memory management**. They show that it represents a vulnerable shared resource through which an attacker could leak sensitive information by showcasing a series of simple attacks on modern runtimes, in particular Java sequential and parallel garbage collection implementations.

JavaScript is a multiparadigm language widely used in the context of web development which, like many other languages designed for this purpose, gives more importance to flexibility and simplicity of use than to correctness guarantees. It includes an *eval* function that can execute statements provided as strings at run-time. Languages implementing this functionality are usually called *dynamic languages*, and tracking information flow in this setting is an intricate yet important problem, as will be explained later on. The large number of features available in JavaScript led the authors of [14] to the identification of a core subset of the language, in or-

der to present a dynamic type system guaranteeing information flow security. This core includes object orientation, higher-order functions, exceptions and dynamic evaluation of code. This work addresses a major issue in the usage of JavaScript, that is client side script inclusion, an extensively used technique for service composition in modern web applications. Included scripts are embedded in the top level of a web page, often in the same page used for the authentication of a user. With user credentials at their disposal, integrated client scripts have unrestricted power to engage interaction with the hosting service. Most browsers today enforce the Same Origin Policy (SOP) which is intended to restrict access for scripts coming from different domains. The SOP offers two alternatives when including a script: either the script is completely isolated, or it is fully integrated. The authors focus on the identification of a tight yet secure integration for scenarios in which these alternatives are not fitting.

Chapter 3

Web Security

In the following sections, several state of the art topics related to Web Security are discussed, and a brief introduction to challenges in the context of Internet of Things (IoT) security is given. After presenting the concept of Web Session Integrity as formalized in [18], the specific scenario of Cross Site Request Forgery (CSRF) attacks will be analysed. Afterwards, several issues related to Security for IoT devices and applications that were addressed during my internship are going to be discussed, along with approaches aiming to achieve security properties by exploiting different techniques.

3.1 Web Session Integrity

The complexity of the web makes it necessary to take into account a large number of variables when discussing the security of a web application. Simple web applications still suffer from a great number of vulnerabilities, which could lead to critical issues if not prevented correctly. Let us consider, for simplicity, a web session as a series of actions by a user on an individual web application, usually within a specific time frame. The integrity or confidentiality of a web

session may be broken at many layers and by a large number of attackers, as highlighted by [4]. In this work, the authors present the most common attacks against web sessions, among which we can identify attacks targeting honest web-browser users establishing an authenticated session with a trusted web application. The authors also review and evaluate existing security solutions and countermeasures that are able to prevent or mitigate such situations. In [6] the authors introduce several black-box testing strategies that aim to detect possible flaws in the implementation of web sessions. After releasing these tactics in a browser extension, they use the latter to assess the security of popular websites and manage to expose a large number of vulnerabilities.

Quite often, web services who need to implement several features or to track the user identity across multiple requests will rely on cookies. Cookies are key-value pairs generated by a server and sent in response to browser requests. After receiving them the browser automatically attaches them to later requests sent to the same website. This behaviour allows for the user not having to re-authenticate whenever he performs an action, for instance. The need for cookies originates from the stateless nature of the HTTP protocol, which implies that every request is going to be executed independently, without any knowledge of requests that were executed before. Through cookies a server is able to discriminate between incoming requests, understanding to which session they belong to.

In the following sections we consider only sessions built on top of HTTP cookies, without considering variants that do not require their use, such as the exchange of JSON Web Tokens through Ajax. This choice does not influence the generality of our discussion, since the cookie based approach still covers the majority of web sessions. Furthermore, we are especially interested in user-authenticated sessions established upon a successful login, where some cookies are used in order to authenticate the user.

3.1.1 Session Hijacking

The fact that cookies represent the only proof of a user's identity makes a web session vulnerable to Session Hijacking attacks. In these attacks a browser run by a given user sends request associated to the identity of the attacker. Despite the numerous threat models under which these attacks were studied and the robust countermeasures available, web developers often ignore the recommended security practices. The adoption of HTTP Strict Transport Security (HSTS) is still not common and cookie security attributes, such as HttpOnly and Secure, are usually unset. The former makes impossible to access the cookies programmatically (i.e. via JavaScript code), while the latter provides confidentiality guarantees on the cookies exchange by constraining the client to only send them over HTTPS connections. Usually, a number $n > 1$ of cookies is used to implement different features in a session. In this situation there might be actually n sub sessions running at the same website, where each cookie is used to retrieve part of the state information related to the session. Sub-Session Hijacking is an attack in which the ideal view of the existence of a single unique user session is broken. An attacker can selectively hijack m sub-sessions, with $m < n$, reducing the security of the whole session to the security of its weakest sub-session. In [5] the authors identify a general overview of Sub-Session Hijacking attacks and introduce a Sub-Session linking technique as a possible countermeasure. They also present a server-side proxy program which enforces this technique on incoming HTTP Requests.

3.1.2 CSRF attacks

In a Cross Site Request Forgery (CSRF) attack, an authenticated user is tricked into performing a security sensitive action against its consent. This attack specifically targets state-changing requests, not theft of data. This is due to the fact that

the attacker has no way to see the response to the forged request. With the help of social engineering, an attacker may trick the users of a web application into executing actions on his behalf. If the victim is a normal user, a successful CSRF attack can force the user to perform requests like transferring funds, changing their email address and similar. If the victim is an administrative account, CSRF attacks can compromise the entire web application. Suppose an authenticated client executes the code received from a malicious server: if this code performs a request to the website he logged onto, the browser will send the login cookie along with it, making so that it will be considered authenticated. A graphical representation of the attack is shown in the following UML sequence diagram.

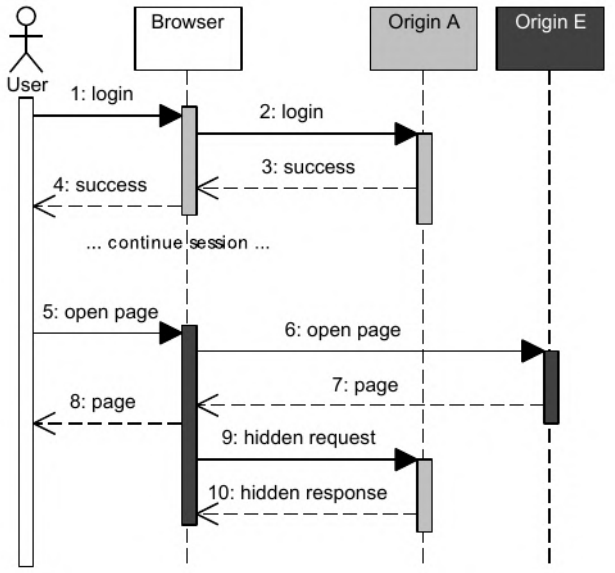

Figure 3.1: UML sequence diagram of a CSRF attack. [18]

To prevent CSRF attacks, web developers have to implement protection mechanisms to filter out malicious cross-site requests. To this end, several techniques are available. Common countermeasures include the usage of custom headers

through JavaScript or specific Anti-CSRF Tokens, embedded in the structure of the web page. Nonetheless, most of these strategies still include several shortcomings.

In [18] the authors describe the development of Login History Dependent (LHD) noninterference, a variant of noninterference which is able to capture the peculiarities and complexities of Web Session Integrity. A significant component in this definition is the presence of a Login History lattice, whose elements represent the integrity levels of an authenticated communication with a domain. During the evaluation of a program the shape of the Login History lattice is altered in response to the occurrence of certain events. For example, when an authentication on domain d is detected, the corresponding integrity level d is added to the Login History. The integrity of communication with that domain can therefore be taken into account thanks to this update, which correctly represents the behaviour of a web browser. LHD noninterference is defined in terms of LHD similarity, as mentioned before with respect to standard noninterference, which takes into consideration also the login history L. The authors show then how this property can be translated to Web Session Integrity and is able to prevent several categories of web security attacks, including CSRF.

3.2 Internet of Things

The Internet of Things (IoT) is a system of related computing devices, in which mechanical and digital machines are provided with unique identifiers. These devices generally have the ability to transfer data over a network, to monitor the environment around them through sensors and to interact with it by means of actuators. This definition has evolved during the years, due to the overlap of multiple technologies such as real-time analytics, machine learning, commodity sensors

and embedded systems technology. Cisco Systems [21] defined the IoT as "simply the point in time when more 'things or objects' were connected to the Internet than people" and estimated that it was "born" between 2008 and 2009, with the things/people ratio growing from 0.08 in 2003 to 1.84 in 2010.

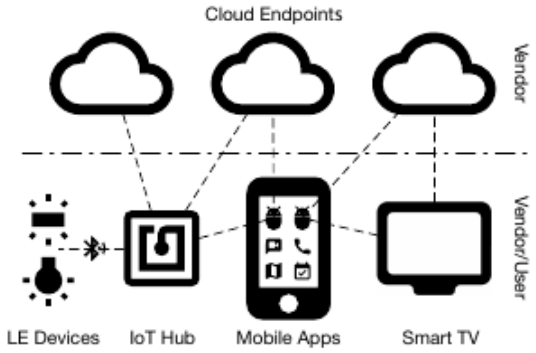

Figure 3.2: Typical home-based IoT setup. [1]

IoT technology is commonly known for products related to the concept of "smart homes", covering devices such as cameras, security systems, heating controllers and similar. These devices are usually related to one ecosystem and are controlled through devices such as smartphones and Vocal Personal Assistants (VPA).Actuators allow the devices to collect information from the environment, leading to the detection of events to which the device is programmed to react. Sensors allow for a concrete interaction with the ecosystem which includes temperature regulation, fire prevention, alarm systems and many more.

3.2.1 Security Issues

Several concerns related to security and privacy of IoT deployments are growing, especially now that a large portion of IoT devices includes vehicles and wearable devices that usually feature remote monitoring capabilities. IoT devices also

have access to new areas of data and many internet-connected appliances were said to be used by companies to spy in people's homes. In [26] the authors report the first security analysis on the ecosystems of Vocal Personal Assistants services. This study led to the discovery of several security weaknesses in their development, which enabled remote attacks from untrusted third parties. Security issues were found even in the simplest objects, including child toys. This has been reported in [9], where the authors analyse three commercially available products exposing several vulnerabilities. They conclude that this situation is indicative of a disconnect between many IoT developers and security best practices.

As mentioned before, another area where concerns are arising is automotive. Modern automobiles are computerized more than ever and potentially vulnerable to several attacks. Devices controlled by computing systems in automobiles (such as brakes, engine, locks and dashboard) have been shown to be vulnerable to attackers who have access to the on-board network. In [8] the authors have analyzes the external attack surface of a modern automobile. They discovered that remote exploitation is feasible via a broad range of attack vectors and that wireless communications channels allow for long distance vehicle control, location tracking, in-cabin audio exfiltration and theft.

Most of the technical security concerns related to IoT devices include weak authentication, forgetting to change default credentials, unencrypted messages sent between devices, SQL injections and poor handling of security updates. However, many IoT devices have limitations regarding the computational power available to them. This constraint can make them unable to directly use basic security measures such as implementing firewalls or using strong cryptosystems to encrypt their communications with other devices.

Different approaches to provide security and privacy guarantees in the Iot have been proposed, exploiting static or dynamic techniques. As examples, we can cite

Soteria [7], a static analysis system for safety validation of both a single IoT application and the whole IoT environment, and HoMonit, [27] a runtime monitor of the wireless traffic designed to detect anomalies and malicious behaviour in smart home apps trying to leak data or spoof events. Nevertheless, the IoT remains unprovided of valid and effective techniques that allow to guarantee the security and privacy of its users. Furthermore, it is desirable that the strategies that will be proposed are based on formally verifiable techniques. For this purpose, a complete analysis of the security implications of these systems is necessary as well as a joint effort between manufacturers and researchers.

Chapter 4

Session Types

In this chapter we will describe the theory of session types, starting from the simplest possible scenario up to the most general one. Writing correct concurrent and distributed code requires effective tools for reasoning about communication protocols. While data types provide an effective tool for reasoning about the shape of exchanged data, communication protocols also require reasoning about the order in which messages are transmitted. It is therefore desirable to have an abstract description of these protocols, in order to be able to analyse them and prevent possible deadlocks and errors in data transmission.

Session types are a type theory focusing on the description and validation of communication protocols. They can be seen as shared agreements between participants in a conversation, specifying the sequence, the direction and the payload of the messages exchanged between them.

Firstly introduced by Honda et al. for binary protocols [15], session types were later generalised to multiparty protocols [16]. Session type theory allows for modelling and verification of binary and multiparty protocols through the analysis of two categories of types which describe different perspectives of the protocol: global types and local types. Global types describe the overall protocol, while

local types describe the contribution of individual participants to the protocol.

For a protocol to be correct, *compatibility* between the local types of its various endpoints is needed. This property, called *duality* in the binary setting, captures the matching between the interaction patterns of the various participants.

Local types can be generated through a projection algorithm from the global type. Compliance between an endpoint program and a local type can then be used to guarantee adherence to the protocol specified by the global type. Two local types obtained as projections from the same global type are compatible by construction. If the protocol is binary, each of the (dual) local types of the participants contains already all the information about the protocol. In this case, the global type of the protocol can be reconstructed from the local types of the participants.

4.1 Binary Session Types

A session is the embodiment of a protocol, whose roles are assigned to a given set of participants. A Binary Session is the simplest possible scenario for a session, involving only two participants. Each session can be labelled with a type representing the usage of communication channels during the interaction. Let us consider the following message sequence diagram, which is a graphical representation of the Buyer Seller protocol, a communication protocol commonly used to introduce Session Types (all the examples in this section are taken from [16]):

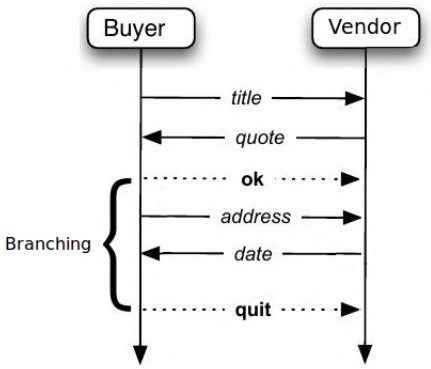

Figure 4.1: Message sequence diagram for the Buyer Seller protocol [16].

In this protocol we have a Buyer entity, which, after establishing a session with a Vendor entity, interacts with it in order to acquire a book. The protocol proceeds as follows:

- At first, the Buyer sends to the Vendor the title of the book in which it is interested.
- The Vendor replies to the client with the amount of money required, and waits for a response from the Buyer, which has now to make a choice between two different communication behaviours:
  - If the Buyer chooses the branch labelled with “ok”:
    - it will send its address to the Vendor;
    - it will receive from the Vendor the expected date of arrival of the book.
  - If the Buyer selects the branch labelled with “quit”, the session ends.

In a binary protocol as the above, there is no need to distinguish between local and global types, since all the information we need is specified in the single type of

one of the participants. Using σ to range over basic data types and p, q, r to denote participants, we can present the syntax for Binary Session Types as follows:

$$
\begin{array}{rcll}
T & ::= & ! \langle q, \sigma \rangle ; T & \text{send} \\
& | & ? (p, \sigma); T & \text{receive} \\
& | & \oplus \langle q, \{ l_i \langle \sigma_i \rangle : T_i \}_{i \in I} \rangle & \text{select} \\
& | & \& (p, \{ l_i \langle \sigma_i \rangle : T_i \}_{i \in I}) & \text{branch} \\
& | & \mu \mathbf{t}.T \mid \mathbf{t} \mid end &
\end{array}
$$

The type $! \langle q, \sigma \rangle ; T$ denotes the action of sending a value of type σ to participant q, followed by the behaviour specified by type T. Correspondingly, the type $? (p, \sigma); T$ represents the action of receiving a value of type σ from participant p, before proceeding as specified by type T. The type $\oplus \langle q, \{ l_i \langle \sigma_i \rangle : T_i \}_{i \in I}$ denotes the selection of any label l_j in the set of labels $\{ l_i | i \in I \}$ offered by q, followed by the behaviour specified by T_j. The type $\& (p, \{ l_i \langle \sigma_i \rangle : T_i \}_{i \in I})$ expresses an offer to participant p of any label l_j in $\{ l_i | i \in I \}$, followed by the behaviour specified by T_j. It is assumed that all the labels l_i are different. Type $\mu \mathbf{t}.G$ allows for the description of recursive protocols.

The select and branch types allow for representation of **branching** in the execution flow, that is the possibility for the protocol to evolve into a **finite number of paths**, each identified by a different label.

We can now specify the type for Buyer:

Buyer :

$$! \langle V, String \rangle ; ? (V, Real); \oplus \langle V, \{ ok : ! \langle V, String \rangle ; ? (V, Int); end, \ quit : end \} \rangle$$

Since the interaction between the two participants is perfectly specular, to ob-

tain the type for Vendor we can simply reverse the type operators, obtaining:

Vendor :

$$?(B, String); !\langle B, Real\rangle; \&(B, \{ok : ?\langle B, String\rangle; !\langle B, Int\rangle; end,\ quit : end\})$$

This example highlights the only requirement for ensuring communication safety in Binary Sessions, which is *duality* between the types of participants. As mentioned above, the whole communication protocol can be described by the type of any of its two participants.

4.2 Multiparty Session Types

A session is said to be Multiparty when it involves more than two participants. Binary session types are not able to represent protocols with more than two parties. Let us consider a variation of the previous example, in which there are two Buyer entities collaborating to acquire a single book:

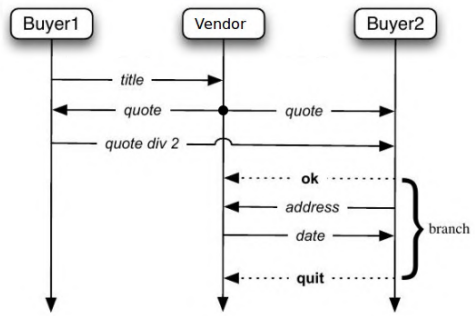

Figure 4.2: Message sequence diagram for the Two Buyers protocol.[16]

The protocol articulates as follows:

- Buyer1 sends a book title to Vendor;

- Vendor sends back a quote to Buyer1 and Buyer2;
- Buyer1 now sends a message to Buyer2, to indicate that it is willing to contribute half of the expense;
- Buyer2 is now in charge of concluding the interaction with the Vendor. It has to make a choice:
  - If Buyer2 chooses the branch labelled with "ok":
    - it will send its address to the Vendor;
    - it will receive from the Vendor the expected arrival date of the book.
  - If Buyer2 selects the branch labelled with "quit", the session ends.

This protocol exposes a limitation of Binary Session Types. Since they can describe protocols with multiple participants only by using multiple binary protocols, we would have to define a binary session type for each pair of participants. Besides not being scalable, this solution would lead to a loss of information about the ordering of messages, since the information about the specific interleaving of binary sessions required by the multiparty interaction would not be capured by the individual session types.

There is therefore a need for a global description of the protocol, which specifies the ordering between these messages, but it would not be a scalable solution to provide each one of the participants with the whole description, since it could account for many operations in which they are not involved. From the notion of **Visibility**, informally meaning that an action is visible to a participant if it is involved in it, we may derive two perspectives of a protocol: **Global** or **Local**. Global Types describe the overall protocol behaviour, while Local Types are specific to each participant and represent only the actions visible to them.

The syntax for global types is the following, assuming $p \neq q$ and $l_i \neq l_j$ for $i \neq j$: :

Global Types	G	$::= \quad p \to q : (\sigma); G'$	interaction
		$\mid \quad p \to q : \{l_i\langle\sigma_i\rangle : G_i\}_{i \in I}$	branching
		$\mid \quad G_1 \parallel G_2$	parallel
		$\mid \quad \mu\mathbf{t}.G$	recursion
		$\mid \quad \mathbf{t}$	variable
		$\mid \quad end$	end

Type $p \to q : (\sigma); G'$ denotes the sending of a message with payload σ from participant p to q, followed by the behaviour specified in G'. Usually $p \neq q$ is assumed since reflexive interaction is not considered. Type $p \to q : \{l_i\langle\sigma_i\rangle : G_i\}_{i \in i}$ represents branching offered from participant q to p between the alternatives in $\{l_i | i \in I\}$. Type $G_1 \parallel G_2$ stands for parallel composition of interactions described in G_1 and G_2, while $\mu\mathbf{t}.G$ is a recursive type to allow for reiterate interactions.

The syntax for local types is the following, assuming $l_i \neq l_j$ for $i \neq j$:

Local Types	T	$::= \quad ! \langle q, \sigma\rangle; T$	send
		$\mid \quad ? (p, \sigma); T$	receive
		$\mid \quad \oplus \langle q, \{l_i\langle\sigma_i\rangle : T_i\}_{i \in I}\rangle$	select
		$\mid \quad \& (p, \{l_i\langle\sigma_i\rangle : T_i\}_{i \in I})$	branch
		$\mid \quad \mu\mathbf{t}.T \mid \mathbf{t} \mid end$	

It should be noted that in Local Types there is no parallel composition as they refer to a single process. Note also that the syntax of local types is the same as the syntax of binary session types. Local Types can be generated through a projection algorithm from the Global type, which also checks several conditions to ensure the well-formedness of the protocol, as we will explain in the following section.

The global Type associated with the Two Buyer Protocol is the following, where $B1$ stands for Buyer1, $B2$ for Buyer2 and V for Vendor:

$$
\begin{aligned}
&B1 \to V : \quad (String)\\
&V \to B1 : \quad (Real)\\
&V \to B2 : \quad (Real)\\
&B1 \to B2 : \quad (Real)\\
&B2 \to V : \{\\
&\qquad ok : B2 \to V : (String)\\
&\qquad\quad V \to B2 : (Int)\\
&\qquad\quad end\\
&\qquad quit : end\\
&\qquad \}
\end{aligned}
$$

Using the **projection** algorithm we obtain the following Local Types for the participants:

Buyer1 : $!\langle V, String\rangle; ?(V, Real); !\langle B2, Real\rangle; end$

Buyer2 : $?(V, Real); ?(B1, Real); \oplus\langle V, \{ok : V!String; V?Int; end,\ quit : end\}\rangle$

Vendor : $?(B1, String); !\langle B1, Real\rangle; !\langle B2, Real\rangle;$

$\&(B2, \{ok : B2?String; B2!Int; end,\ quit : end\})$

4.3 Projection of a Multiparty Protocol

The following assumptions are made on the transmission of messages.

- **Asynchrony:** a sender should not have to wait for a message to be sent before continuing its computation. Sending actions are nonblocking. How-

ever, the order of messages between a given pair of participants is preserved. The allowed message permutations are expressed by the following equations, where we assume $p \neq p'$:

$$!\langle p, \sigma\rangle; !\langle p', \sigma'\rangle; end \approx !\langle p', \sigma'\rangle; !\langle p, \sigma\rangle; end$$

$$\oplus\langle p, \{l_i : \oplus\langle p', \{l'_j : T_{ij}\}_{j\in J}\rangle\}_{i\in I}\rangle \approx \oplus\langle p'\{l'_j : \oplus\langle p, \{l_i : T_{ij}\}_{i\in I}\rangle\}_{j\in J}\rangle$$

The first equation allows permutation of two consecutive outputs directed to different participants. The second equation allows permutation of two consecutive selections directed to different participants.

- **Order preservation:** the messages between a participant p and a participant q are received in the same order in which they have been sent.
- **Reliability:** a message is never lost nor blocked during transmission.

When the **projection function** given below is defined for a global type, it is possible to obtain the local types for each of its participant. These maintain dependencies and operations concerning a specific participant and exclude the others. The projection function is a partial function because of the condition $\forall i, j \in I, G_i \upharpoonright p = G_j \upharpoonright p$, and because of the condition of disjointnesses for the sets of participants of G_1 and G_2 in $(G_1 \parallel G_2)$.
The projection of G onto p, written $G \upharpoonright p$, is inductively defined as follows:

- $(p \to q : \langle\sigma\rangle.G') \upharpoonright r =$

$$\begin{cases} !\langle q, \sigma\rangle.(G' \upharpoonright r) & if\ r = p \\ ?(p, \sigma).(G' \upharpoonright r) & if\ r = q \\ (G' \upharpoonright r) & if\ r \neq p\ and\ r \neq q \end{cases}$$

- $(p \to q : \{l_j : G_j\}_{j \in J}) \upharpoonright r =$

$$\begin{cases} \oplus\langle q, \{l_j : (G_j \upharpoonright r)\}_{j \in J}\rangle & if\ r = p \\ \&(p, \{l_j : (G_j \upharpoonright r)\}_{j \in J}), & if\ r = q \\ (G_i \upharpoonright r) & if\ r \neq p\ and\ r \neq q \\ & and\ \forall i, j \in I, G_i \upharpoonright r = G_j \upharpoonright r \end{cases}$$

- $(G_1, G_2) \upharpoonright r =$

$$\begin{cases} G_i \upharpoonright r & if\ r \in G_i\ and\ r \notin G_j, i \neq j \in \{1, 2\} \\ end & if\ r \notin G_1\ and\ r \notin G_2 \end{cases}$$

- $(\mu \mathbf{t}.G) \upharpoonright r = \mu \mathbf{t}.(G \upharpoonright r), \mathbf{t} \upharpoonright r = \mathbf{t}\ and\ end \upharpoonright r = end$

- when some of the side conditions do not hold, the map is undefined..

It is worth noting that in the branching case, all projections onto participants different from p and q have to generate an identical local type. In the case of parallel composition, participant p should not be contained in more than one type, in order to ensure that every type is related to a single thread of execution.
If a global type is projectable, then the local types for its participants can be generated. These local types can then be used to perform type checking on the processes implementing the participants.

4.4 Safety Properties

The main properties ensured by Session Types are:

- **Communication safety:** no communication errors can occur in a session.

- **Session Fidelity:** the communication sequence in a session follows the scenario declared in the global type.
- **Progress:** every sent message is going to be received, every process waiting for a message will receive it.

Communication safety and session fidelity rely on the Subject Reduction property of the type system. To give an intuition of how a session type system manages to ensure these properties, we introduce now a simple process calculus, where single sessions (which we assume to be already initiated) are represented as networks of sequential processes. Processes are ranged over by P, Q, networks by $\mathbf{N}, \mathbf{N}'$ and expressions by e, e'. As for types, participants are denoted by p, q, r.

The syntax for finite processes is the following (for simplicity we do not consider recursive processes):

$$
\begin{array}{lrll}
\text{Process} & P & ::= & \langle q, e\rangle.P & \text{Value sending} \\
& & \mid & (p, x).P & \text{Value reception} \\
& & \mid & \oplus\langle q, l\rangle.P & \text{Selection} \\
& & \mid & \&(p, \{l_i : P_i\}_{i\in I}) & \text{Branching} \\
& & \mid & \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q & \text{Conditional} \\
& & \mid & \mathbf{0} & \text{Inaction}
\end{array}
$$

Multiparty sessions are described as asynchronous networks of processes enclosed into participants. We denote by $p[\![\, P \,]\!]$ the process P enclosed into particpant p. The intuition is that $p[\![\, P \,]\!]$ represents participant p executing process P. Then, a network is a parallel composition of components $p[\![\, P \,]\!]$, where all p's are assumed to be different.

Definition (Networks)**.** *Networks are defined by:*

$$\mathbf{N} ::= p[\![\, P \,]\!] \mid \mathbf{N} \parallel \mathbf{N}' \qquad \mathsf{Part}(\mathbf{N}) \cap \mathsf{Part}(\mathbf{N}') = \emptyset$$

where $\mathsf{Part}(\mathbf{N})$, the set of participants of $\mathbf{N}$, is defined by:

$$\begin{aligned} \mathsf{Part}(\mathsf{p}[\![\, \mathsf{P} \,]\!]) &= \{p\} \\ \mathsf{Part}(\mathbf{N} \parallel \mathbf{N}') &= \mathsf{Part}(\mathbf{N}) \cup \mathsf{Part}(\mathbf{N}') \end{aligned}$$

The operator $\parallel$ is assumed to be associative and commutative, with neutral element $p[\![\, \mathbf{0} \,]\!]$ for each p. These laws yield the structural congruence for networks.

Since communication is assumed to be asynchronous, as discussed earlier, sessions should be asynchronous networks of processes, that is, networks executing in parallel with queues. Formally, *messages* and *queues* are defined as follows:

$$\begin{array}{llcll} \text{Messages} & m & ::= & (p, q, v) & \text{Enqueued Value} \\ & & \mid & (p, q, l) & \text{Enqueued Label} \end{array}$$

$$\begin{array}{llcll} \text{Queue} & h & ::= & h \cdot m & \text{Enqueueing} \\ & & \mid & \emptyset & \end{array}$$

We can now define asynchronous networks:

Definition (Asynchronous Networks)**.** *Asynchronous networks are defined by:*

$$\mathbf{N}_A ::= \mathbf{N} \parallel h$$

The operational semantics for our calculus is defined by the following set of reduction rules, where $P\{x \mapsto v\}$ represents process P where value v is assigned to variable x and $e \downarrow v$ means that expression e evaluates to value v.

$$
\begin{array}{lcll}
p[\![\, !\langle q, e\rangle.P \,]\!] \parallel \mathbf{N} \parallel h & \rightarrow & p[\![\, P \,]\!] \parallel \mathbf{N} \parallel h \cdot (p, q, v) \ \text{ if } \ e \downarrow v & \text{[Send]} \\
q[\![\, ?(p, x).Q \,]\!] \parallel \mathbf{N} \parallel (p, q, v) \cdot h & \rightarrow & q[\![\, Q \,]\!]\{x \mapsto v\} \parallel \mathbf{N} \parallel h & \text{[Rcv]} \\
p[\![\, \oplus\langle q, l\rangle.P \,]\!] \parallel \mathbf{N} \parallel h & \rightarrow & p[\![\, P \,]\!] \parallel \mathbf{N} \parallel h \cdot (p, q, l) & \text{[Sel]} \\
q[\![\, \&(p, \{l_i : Q_i\}_{i \in I}) \,]\!] \parallel \mathbf{N} \parallel (p, q, l_j) \cdot h & \rightarrow & q[\![\, Q_j \,]\!] \parallel \mathbf{N} \parallel h \ \text{ if } \ j \in I & \text{[Branch]} \\
p[\![\, \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \,]\!] \parallel \mathbf{N} \parallel h & \rightarrow & p[\![\, P \,]\!] \parallel \mathbf{N} \parallel h \ \text{ if } \ e \downarrow true & \text{[If-T]} \\
p[\![\, \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \,]\!] \parallel \mathbf{N} \parallel h & \rightarrow & p[\![\, Q \,]\!] \parallel \mathbf{N} \parallel h \ \text{ if } \ e \downarrow false & \text{[If-F]}
\end{array}
$$

We now introduce the session type system for processes and networks. We denote by Γ the standard environment which associates basic data types to variables.

Typing judgments for expressions, processes and networks have the following shape:

$$\Gamma \vdash e : \sigma \ \text{ and } \ \Gamma \vdash P : T \ \text{ and } \ \Gamma \vdash \mathbf{N} : G$$

The typing rules for processes are the following:

$$\textsc{Nil}\qquad \Gamma \vdash \mathbf{0} : end$$

$$\textsc{Send}\quad \frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash P : T}{\Gamma \vdash\ !\langle q, e\rangle.P\ :!\ \langle q, \sigma\rangle; T} \qquad \textsc{Rcv}\quad \frac{\Gamma, x : \sigma \vdash P : T}{\Gamma \vdash\ ?(p, x).P\ :?\ (p, \sigma); T}$$

$$\textsc{Sel}\quad \frac{l = l_j \qquad \Gamma \vdash P : T_j \qquad j \in J}{\Gamma \vdash \oplus\langle q, l\rangle.P\ :\ \oplus\ \langle q, \{l_i : T_i\}_{i\in I}\rangle}$$

$$\textsc{Branch}\quad \frac{\Gamma \vdash P_i : T_i \qquad \forall i \in I}{\Gamma \vdash \&(p, \{l_i : P_i\}_{i\in I})\ :\ \&\ (p, \{l_i : T_i\}_{i\in I})}$$

$$\textsc{If}\quad \frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash P : T \qquad \Gamma \vdash Q : T}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q : T}$$

The typing rule for networks is:

$$\textsc{Net}\quad \frac{\Gamma \vdash P_i : G \upharpoonright p_i \qquad \mathsf{Part}(\mathsf{G}) \subseteq \{p_i \mid i \in I\}}{\Gamma \vdash \prod_{i\in I} p_i[\![\, P_i \,]\!] : G}$$

Rule SEND verifies that both the sent expression e and the continuation process P are typable and then assigns to the output process the corresponding output type. Rule RCV is similar except that it checks typability of P in the environment Γ extended with the assumption that the input variable has type σ.

Rule SEL verifies that the continuation process P follows the session type of the branch related to the label that is selected. Rule BRANCH verifies the same

condition for all of the branches offered by the branch construct. Rule IF constrains the branches of a conditional to have the same behaviour of interaction with the other parties, since compatibility with the other parties must hold no matter which branch is chosen.
It is interesting to note that all the "prefix rules" [Send], [Rcv], [Sel] and [Branch] consume the prefix of the session type of the process.

The typing rule for asynchronous networks is more involved as it requires typing the queues and projecting them onto participants, as well as concatenating the type of a network with the type of a queue. Moreover, queues must be handled modulo permutation of independent messages, in agreement with the "asynchrony equations" shown in page **??**. We therefore omit it in this presentation.

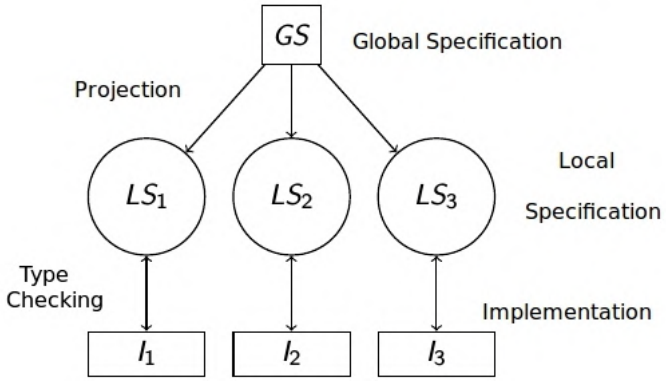

Figure 4.3: Diagram representing the different levels of Session Types.

The diagram above summarises the Session Type approach for the verification of distributed systems. Thanks to local checks, which can be performed statically, Session Types theory allows us to prove properties which are critical in the assesment of the correctness of a distributed system.

Chapter 5

Multitier Programming

In the following sections we will describe the multitier paradigm for programming and delve on two examples of languages adopting it: Hop, developed at INRIA Sophia Antipolis, and Links, a multitier language implementing Session Types. The multitier paradigm aims at solving the problems caused by the client-server architecture of web applications. This asymmetric structure most often obliges programmers to use different languages to implement the whole system. Tipically, a web application is organized in three tiers, each running on a separate computer. The logic, which resides on the middle-tier server, generates web pages. These are sent to a front-end browser, which is generally capable of formulating queries. The latter must traverse the entire structure to reach a back-end database.

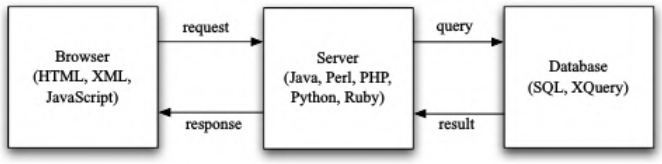

Figure 5.1: Schematics of the three tier model.[10]

As shown in the figure above, the logic of the application could be written in a mixture of several languages, each one of them related to one of the three tiers involved. After learning many of these, a developer faces the problem of finding a way to connect them together. For example, it is necessary to ensure that a HTML form or a SQL query produces the data expected by a Java program. This issue is usually referred to as the impedance mismatch problem. It may be argued that this situation can lead to poor implementation and does not help programmers to focus on security aspects of the application, which can be distributed in all of the tiers mentioned. Multitier programming focuses on being able to program the application as a single code for both server and client. This approach is able to solve the impedance mismatch problem and reduces development complexity.

5.1 Hop

Hop [24] is a multitier language designed for programming interactive web applications, which adopts a programming model based on two computation levels. The first level is used to program the server behaviour and for designing graphical user interfaces (GUI). The second level is used for programming animations and interactions of interfaces with the user. This approach allows for a single packaging of the whole application and separates its view and logic. Hop eases the implementation of web application by abstracting many operations required by the web. It supports object-oriented programming, exceptions, modules, multi-threading and provides various tools and libraries. Notably, Hop provides original constructions designed specifically for programming web applications, which consist of functions calls traversing the web and a particular mechanism for event notification. This is due to the fact that a Hop program is executed on multiple engines residing on different machines. Hop allows the server's and

client's execution flows to communicate one with each other by means of this peculiar function calls and event management.

5.1.1 A dual language

By promoting a computation model in which the main tasks are executed on the server, while the GUI is executed on clients, Hop applications are scalable by construction. A Hop program has to be uploaded on a Hop server, conforming to the HTTP protocol, in order to be binded to a provided URL. This is the one that will be used by the client's web browsers to start the application, allowing for the server and client to interact one with each other. A Hop program, as mentioned before, is executed on several engines (namely physical computers) at the same time. The main engine resides in the server and is dedicated to executing the logic of the program. It focuses on computations which require a high CPU usage and manages privileged operations, such as file access and alteration. The client engine is dedicated to the GUI and its related animations and functionalities.

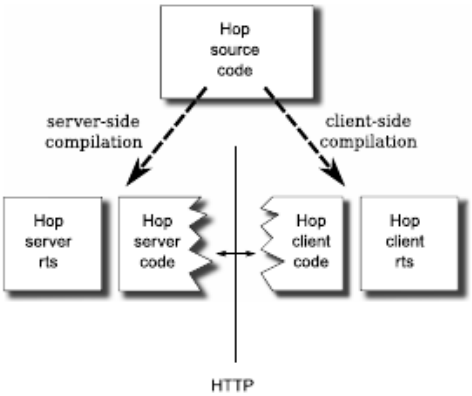

Figure 5.2: Architecture of a Hop program.[24]

Once started, the program is executed first on the main engine, in order to

elaborate a description of the GUI. Once this is sent to the client engin, the execution starts to flow from the latter to the former, and vice versa. Hop also allows for more execution flows to run in parallel.

5.1.2 Communication and service calls

Interactions between the two engines are enabled by remote function calls and event loops. These features allows for a practical implementation of application which frequently need to communicate. The client engine (or GUI engine) can invoke functions, namely services, from the main engine. The main engine is able to notify events to the GUI engines. We will focus now on service calls since they are going to play a relevant role in the remaining chapters of the document.
When programming in Hop, on can considers HTML's URLs to be very similar to functions in programming languages. For example, the URL `https://www.ecosia.org` may be considered as a function named www.ecosia.org, whose signature is: **unit** $\rightarrow$ **HTMLtree**.
One click of the user on the link provided by the `<a>` markup is sufficient to call this function. Similarly, a different function is denoted by the URL `https://www.ecosia.org/search`. This function is called as a response to the user click on the "search" icon. Since this function takes a parameter named q as its argument, namely the text which is object of the research, we can define its signature as: **String** $\rightarrow$ **HTMLtree**.

In Hop, every function is linked to special functions, called services. Services inhabit the server and can be called by clients. A service is defined in the following syntax:

```
(define-service (<ident> <ident0> ...)
        <expression> )
```

`<ident>` is the name of the service and `<ident0> , ...` are its parameters.The `define-service` construct does not only bind a function to an indentifier residing on the server, it also binds it to a URL, which can be used to run a Hop program. Service calls from the client to the server can be used to produce either complete web pages or partial results. Any Hop service can be called, from a client, with the following syntax:

```
(with-hop (service arg0, ...)
        [(lambda (h) ...success expression...)
        [(lambda (h) ...failure expression...)]])
```

Through the `with-hop` syntax, a client calls the function `service` with the arguments `arg0,...`. Once the call is completed, two alternative situations may occur. In case of success the success procedure is executed. Otherwise, the failure procedure is executed. Both of them accept as argument the result produced by the service evaluation of the call.

5.2 Links

Links is a multitier functional programming language which aims at easing the impedance mismatch problem. It provides a language which allows for the specification of the behaviours of all tiers involved in a web application. To do so, it generates code for each tier by translating parts of the program into JavaScript code, which can be run by the the browser, other parts into bytecode, which can be run by the server, and the remaining bits into SQL for the database. Namely, as reported in [10], the server component of Links is composed of a static analyzer, a translator to JavaScript and SQL and an interpreter for Links code. The client component compiles to JavaScript, providing support for Ajax. Links adopts a communication system based on the exchange of messages to provide support for

distributed applications, such as Erlang, and many other features incorporated by other functional programming languages.

As in Hop, a Links program is distributed and executed across two locations, a client and a server. Keywords `client` and `server` can be used to annotate function definitions, to indicate where it shoud run. Functions annotated with `client` can call functions with `server`, and the other way around. This is made possible by the usage of a variation of the continuation-passing technique that encapsulates all of the server's sate information inside a parameter passed to the client. Links servers are thus scalable and require little resources.

An interesting feature of Links is to provide support for Session Types. These were added to the language by Lindley and Morris in [20] by means of row polymorphism and lightweight linear typing. Lindley and Morris base themselves on a language introduced in [19], called GV. GV is a functional language with Session Types that was proven to be deadlock free, deterministic and terminating. In [20] GV is extended with polymorphism, row types, subkinding. This allows for the integration of linear types, which are very useful to introduce Session Types. The resulting language is more expressive, while retaining the same safety property. Since version 0.6, Links includes an extension based on this language. Session types are checked by means of static analysis, allowing for the development of the application to be driven by the communication protocol it is supposed to comply to. However, most accounts of session typing do not account for failure. This limits their use in most web applications, where failure is pervasive and needs to be handled carefully. To solve this problem, in [12] GV and, hence, Links are extended further to provide for exception handling. In order to concretze this, exceptions are embedded into Session Types theory for GV by incorporating the approach of [22] leveraging on linear types with explicit cancellation.

Chapter 6

Webi

In this section, we introduce the Webi semantic framework, as proposed by Tamara Rezk and her colleagues. In the next section we will define our specific version of Webi, tailored towards the representation of Authenticated Sessions. The Webi semantics is meant to represent the interactions occurring on the web between different servers and many clients which are able to browse multiple web-sites at the same time. The semantics is parametric on client and server transition relations $\rightsquigarrow_S$ and $\rightsquigarrow_C$ (representing the evaluation of their code). Moreover, it exhibits one of the most prominent features of a web application, that is the transfer of code and values from a server to a client. The first time a client connects to a server it will answer with some code to be executed. This code might call other servers as well, or simply allow for interaction between server and client.

6.1 Client and Server languages

6.1.1 Client Language

The programs used to instantiate client configurations have the following syntax:

$$
\begin{aligned}
Q \;::=\; & x := e \mid Q; Q \mid \mathsf{skip} \mid \mathsf{return}\ e \\
& \mid \mathsf{if}\ e\ \mathsf{then}\ Q\ \mathsf{else}\ Q \mid \mathsf{while}\ e\ \mathsf{do}\ Q \\
& \mid \mathsf{branch}(l)\ \{l_1 : Q_1\ \ldots l_n : Q_n\} \\
& \mid \mathsf{call}\ u\ \mathsf{with}\ \overrightarrow{x} := e\ \mathsf{then\ do}\ \lambda x.Q \\
e \;::=\; & v \mid x \mid op(e_1, \ldots, e_n) \\
v \;::=\; & \mathsf{undefined} \mid \mathsf{true} \mid \mathsf{false} \mid 1 \mid 2 \mid \ldots
\end{aligned}
$$

We let v range over values and require that the set of values, called Val, should contain a special value named undefined. We let μ range over memories, that is, finite functions from variables to values. We write $\mu\{x \mapsto v\}$ for the memory that maps x to v and any $y \neq x$ to $\mu(y)$. In Figure 1 we present the small-step operational semantics for the Client language. This semantics is given in terms of two transition relations: an undecorated transition relation $\rightsquigarrow_C$ (also called reduction relation), and a *decorated* transition relation $\rightsquigarrow_C^{\delta}$, where δ is a *transition decoration*, defined by:

$$\delta ::= u?\overrightarrow{x} = \overrightarrow{v}, \lambda x.Q$$

These record some relevant information about service calls. Both transition relations are defined on *client configurations*, which have the form $\langle Q, \mu\rangle$ or $\langle v, \mu\rangle$. The latter, called *final* client configurations, represent termination with return value v and final memory μ. When writing $\langle Q, \mu\rangle$, we implicitly assume that the variables of Q are included in the domain of μ. All expressions are evaluated

through the function $[\![e]\!](\mu)$:

$$[\![e]\!](\mu) = \begin{cases} v & if \quad e = v \\ \mu(x) & if \quad e = x \\ op([\![e_1]\!](\mu) \ldots [\![e_n]\!](\mu)) & if \quad e = op(e_1, \ldots e_n) \end{cases}$$

Figure 6.1: Small-Step Semantics for Client Language.

ASSIGN

$$\frac{[\![e]\!](\mu) = v}{\langle x := e, \mu\rangle \rightsquigarrow_C \langle \mathsf{undefined}, \mu\{x \mapsto v\}\rangle}$$

SKIP

$$\frac{}{\langle \mathsf{skip}, \mu\rangle \rightsquigarrow_C \langle \mathsf{undefined}, \mu\rangle}$$

RETURN

$$\frac{[\![e]\!](\mu) = v}{\langle \mathsf{return}\ e, \mu\rangle \rightsquigarrow_C \langle v, \mu\rangle}$$

SEQR

$$\frac{\langle Q_1, \mu\rangle \rightsquigarrow_C \langle v, \mu'\rangle \qquad v \neq \mathsf{undefined}}{\langle Q_1; Q_2, \mu\rangle \rightsquigarrow_C \langle v, \mu'\rangle}$$

SEQS

$$\frac{\langle Q_1, \mu\rangle \rightsquigarrow_C \langle Q_1', \mu'\rangle}{\langle Q_1; Q_2, \mu\rangle \rightsquigarrow_C \langle Q_1'; Q_2, \mu'\rangle}$$

SEQT

$$\frac{\langle Q_1, \mu\rangle \rightsquigarrow_C \langle \mathsf{undefined}, \mu'\rangle}{\langle Q_1; Q_2, \mu\rangle \rightsquigarrow_C \langle Q_2, \mu'\rangle}$$

SEQSL

$$\frac{\langle Q_1, \mu\rangle \rightsquigarrow_C^{\delta} \langle Q_1', \mu'\rangle}{\langle Q_1; Q_2, \mu\rangle \rightsquigarrow_C^{\delta} \langle Q_1'; Q_2, \mu'\rangle}$$

SEQTL

$$\frac{\langle Q_1, \mu\rangle \rightsquigarrow_C^{\delta} \langle \mathsf{undefined}, \mu'\rangle}{\langle Q_1; Q_2, \mu\rangle \rightsquigarrow_C^{\delta} \langle Q_2, \mu'\rangle}$$

COND

$$\frac{[\![e]\!](\mu) = \alpha \qquad \alpha \in \{\mathsf{true}, \mathsf{false}\}}{\langle \mathsf{if}\ e\ \mathsf{then}\ Q_{\mathsf{true}}\ \mathsf{else}\ Q_{\mathsf{false}}, \mu\rangle \rightsquigarrow_C \langle Q_\alpha, \mu\rangle}$$

WHILETRUE

$$\frac{[\![e]\!](\mu) = \mathsf{true}}{\langle \mathsf{while}\ e\ \mathsf{do}\ Q, \mu\rangle \rightsquigarrow_C \langle Q; \mathsf{while}\ e\ \mathsf{do}\ Q, \mu\rangle}$$

WHILEFALSE

$$\frac{[\![e]\!](\mu) = \mathsf{false}}{\langle \mathsf{while}\ e\ \mathsf{do}\ Q, \mu\rangle \rightsquigarrow_C \langle \mathsf{undefined}, \mu\rangle}$$

BRANCH

$$\frac{l = l_i}{\langle \mathsf{branch}(l)\ \{l_1 : Q_1\ \ldots l_n : Q_n\}, \mu\rangle \rightsquigarrow_C \langle Q_i, \mu\rangle}$$

CALL

$$\frac{[\![e]\!](\mu) = \overrightarrow{v}}{\langle \mathsf{call}\ u\ \mathsf{with}\ \overrightarrow{x} := e\ \mathsf{then\ do}\ \lambda x.Q, \mu\rangle \rightsquigarrow_C^{u?\overrightarrow{x}=\overrightarrow{v},\lambda x.Q} \langle \mathsf{undefined}, \mu\rangle}$$

Rule [Assign] yields a final configuration where the value is undefined and the memory is obtained from μ by updating the entry for the variable x with the value of expression e. Rule [Return] evaluates an expression e to produce a return value v, while leaving the memory unchanged. There are five rules for sequential composition : three specifying the reduction relation, and two specifying the decorated transition relation. Rule [SeqR] applies when the first component of the sequential composition yields a value different from undefined. In this case, since a program is not supposed to continue after returning a proper value, the second component is discarded. Rule [SeqS] applies when the first component evolves to another program, while Rule [SeqT] applies when the first component terminates, i.e. leads to the value undefined : in this case, the second component starts to be executed. Rules [SeqSL] and [SeqTL] are the decorated counterparts of [SeqS] and [SeqT]. Rules [WhileTrue], [WhileFalse] and [Cond] are standard. Rule [Branch] models the capability of a program to offer multiple possible behaviours by associating each one of them to a *label* and running the program Q_i associated to the selected label l_i. Rule [Call] models a client call to service u, with parameter e and continuation program $\lambda x.Q$: expression e is evaluated to a values $\overrightarrow{v}$ to be bound to variable $\overrightarrow{x}$, and both this binding and the continuation function are recorded in the decoration of the transition. This information will then be fetched by the called service as specified by the WEBI semantics described in Section 6.3.

6.1.2 Server Language

The Server Language has a richer syntax for expressions than the Client Language. In addition to the standard expressions e, introduced previously, it includes expressions of the form $\tilde{}t$, called *tilde expressions*, and expressions of the form $\$x$, called *dollar expressions*. Formally, the sets of server expressions and values,

ranged over by $e_\$$ and $v_\sim$, are defined by:

$$\begin{aligned} e_\$ &::= v_\sim \mid x \mid \$x \mid op(e^1_\$, \ldots, e^n_\$) \mid \tilde{}t \\ v_\sim &::= v \mid \tilde{}Q \end{aligned}$$

where t is defined by:

$$\begin{aligned} t ::= {} & x := e_\$ \mid t;t \mid \mathsf{skip} \mid \mathsf{return}\ e_\$ \\ & \mid \mathsf{if}\ e_\$\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{while}\ e_\$\ \mathsf{do}\ t \\ & \mid \mathsf{branch}(l)\ \{l_1 : P_1\ \ldots l_n : P_n\} \\ & \mid \mathsf{call}\ u\ \mathsf{with}\ x := e_\$\ \mathsf{then\ do}\ \lambda x.t \end{aligned}$$

Hence Q is a particular case of t (where all expressions $e_\$$ are simply client expressions e), and expressions of the form $\tilde{}Q$ are considered to be values in the server language.

The syntax for the programs used to instantiate server configurations is the following:

$$\begin{aligned} P ::= {} & x := e_\$ \mid P;P \mid \mathsf{skip} \mid \mathsf{return}\ e_\$ \\ & \mid \mathsf{if}\ e_\$\ \mathsf{then}\ P\ \mathsf{else}\ P \mid \mathsf{while}\ e_\$\ \mathsf{do}\ P \\ & \mid \mathsf{branch}(l)\ \{l_1 : P_1\ \ldots l_n : P_n\} \end{aligned}$$

Note that the syntax of server programs only differs from that of client programs for the absence of the call construct and for the use of expressions $e_\$$ instead of e.

The semantics of the Server Language is defined on *server configurations* of the form $\langle P, \mu\rangle$ or $\langle v_\sim, \mu\rangle$, the latter being called again *final*. We start by defining the evaluation of expressions. Dollar expressions are evaluated to values and tilde

expressions where all dollar expressions have been evaluated are treated as values in the language. The expression evaluation function $\{\!|e_\$|\!\}(\mu)$ uses an auxiliary function $\phi(t, \mu)$. Formally, $\{\!|e_\$|\!\}(\mu)$ is defined by:

$$\{\!|e_\$|\!\}\mu = \begin{cases} v & \text{if } e_\$ = v \\ \mu(x) & \text{if } e_\$ = x \text{ or } e_\$ = \$x \\ op(\{\!|e^1_\$|\!\}, \ldots, \{\!|e^n_\$|\!\}) & \text{if } e_\$ = op(e^1_\$, \ldots, e^n_\$) \\ \tilde{}\phi(t, \mu) & \text{if } e_\$ = \tilde{}t \end{cases}$$

where:

$$\phi(t, \mu) =$$

$$\begin{cases} x := \{\!|e_\$|\!\}(\mu) & \text{if } t = x := e_\$ \\ \phi(t_1, \mu); \phi(t_2, \mu) & \text{if } t = t_1; t_2 \\ \mathsf{if}\ \{\!|e_\$|\!\}(\mu)\ \mathsf{then}\ \phi(t_1, \mu)\ \mathsf{else}\ \phi(t_2, \mu) & \text{if } t = \mathsf{if}\ e_\$\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \\ \mathsf{while}\ \{\!|e_\$|\!\}(\mu)\ \mathsf{do}\ \phi(t, \mu) & \text{if } t = \mathsf{while}\ e_\$\ \mathsf{do}\ t \\ \mathsf{call}\ u\ \mathsf{with}\ x := \{\!|e_\$|\!\}(\mu)\ \mathsf{then\ do}\ \lambda x.\phi(t, \mu) & \text{if } t = \mathsf{call}\ u\ \mathsf{with}\ \overrightarrow{x} := e_\$\ \mathsf{then\ do}\ \lambda x.t \\ t & \text{if } t = \mathsf{skip} \text{ or} \\ & \text{if } t = \mathsf{branch}(l)\ \{l_1 : P_1\ \ldots l_n : P_n\} \end{cases}$$

The small-step operational semantics for server programs is then formally the same as that for client programs, replacing v by $v_\sim$ and $[\![e]\!](\mu)$ by $\{\!|e_\$|\!\}(\mu)$.

6.2 Webi configurations

We now present the Webi semantics, which describes the interaction between clients and servers. The semantics is defined on *Webi configurations*, which are pairs of the form:

$$\langle SS, CC \rangle$$

where SS is a set of *concrete* server configurations ss and CC is a multiset of *concrete* client configurations cc. The possible forms of ss and cc are:

$$\begin{array}{rcl} ss & ::= & \langle \mathsf{service}\ u(\overrightarrow{x})\{P\}, \mu \rangle \mid \\ & & \langle P, \mu \rangle^j \mid \\ & & \langle \langle P, \mu \rangle^k \rangle^j \mid \\ & & \langle v_\sim, \mu \rangle^j \\ \\ s & ::= & \langle P, \mu \rangle \mid \\ & & \langle v_\sim, \mu \rangle \\ \\ cc & ::= & \langle \langle c, B, T \rangle^j \rangle^u \mid \\ & & \langle \langle \mathsf{boot} \rangle^j \rangle^u \mid \\ \\ c & ::= & \langle Q, \mu \rangle \mid \\ & & \langle v, \mu \rangle \end{array}$$

Here s is a server configuration and c is a client configuration, as described previously. In the syntax of ss, u is the name of a service, P is a server program, $\overrightarrow{x}$ is a set of input parameters, $v_\sim$ is a server value and μ is the memory in which the program P is going to be executed. The superscripts j and k are used to specify respectively the session name and the return address for callback functions. In the syntax of cc, B is the callback function queue and T is the thunks queue. Here the superscripts j and u denote respectively the session name and the name of the

service that is the source of the configuration. A callback function represents a subroutine waiting to receive an input from the server in order to be calculated. It is instantiated after a HTTP Request to a server, and has the form $(\lambda x.Q)^k$, where k is the return address for the Server. We will write $B :: (\lambda x.Q)^k$ for enqueueing. When the server returns an input to the callback function, the two are paired and enqueued in the thunks queue. When the current active code terminates, it is substituted with the first of the thunks queue elements. Client code is generated by two different functions: boot maps a server value to client code, while gencc is used to map a server value to a client value.

6.3 Webi Semantics

We present now the semantic rules of Webi. There are two kinds of transition relations here as well, one that is unlabelled and the other that is labelled by decorations Δ, which are richer than the δ's introduced previously, and are given by:

$$\delta ::= \mathsf{boot} \mid u?\overrightarrow{x} = \overrightarrow{v} \mid u?\overrightarrow{x} = \overrightarrow{v}, \lambda x.Q$$

URLREQ

$$\frac{\langle \mathsf{service}\ u(\overrightarrow{x})\{P\}, \mu\rangle \in SS \qquad j\ \mathsf{fresh} \qquad CC' = CC \cup \{\langle\langle \mathsf{boot}\rangle^j\rangle^u\} \qquad SS' = SS \cup \{\langle P, \mu\{\overrightarrow{x} \mapsto \overrightarrow{v}\}\rangle^j\}}{\langle SS, CC\rangle \rightsquigarrow^{u?\overrightarrow{x}=\overrightarrow{v}} \langle SS', CC'\rangle}$$

This rule specifies how a client making a Url Request and the target server u interact. The client initiates a new session j with the server by sending a vector of values $\overrightarrow{v}$ to be associated with the vector of parameters $\overrightarrow{x}$. A new client configuration $\langle\langle \mathsf{boot}\rangle^j\rangle^u$ is added to the client configuration set, which will execute the boot function to generate the client code from the server response. A server con-

figuration is added to the server configuration set to calculate a response for the client. Note that this rule can be applied anytime, as long as SS contains a service declaration $\langle \mathsf{service}\ u(\overrightarrow{x})\{P\}, \mu\rangle$. This represents the fact that an UrlRequest can be issued anytime, whatever the set of client configurations CC.

CLIENTBOOT

$$\frac{SS = SS' \cup \{\langle v_\sim, \mu\rangle^j\} \qquad CC = CC'' \cup \{\langle\langle \mathsf{boot}\rangle^j\rangle^u\} \qquad \mathsf{boot}(v_\sim) = Q \qquad CC' = CC'' \cup \{\langle\langle Q, \epsilon, \epsilon\rangle^j\rangle^u\}}{\langle SS, CC\rangle \rightsquigarrow^{\mathsf{boot}} \langle SS', CC'\rangle}$$

This rule specifies how a client that has to boot may receive code from the server. This happens if there exist a server configuration and a client configuration that are labelled with the same session name j. Note that this implies that the server configuration was created via Rule [UrlReq] by the same service u that appears as a superscript in the client configuration. The boot function returns the client code that becomes the active client code in the new configuration, while $\langle\langle v_\sim, \mu\rangle^j\rangle$ is lost, due to the stateless nature of the HTTP protocol.

CLIENTCALL

$$\frac{CC = CC'' \cup \{\langle\langle c, B, T\rangle^j\rangle^u\} \qquad c \rightsquigarrow_C^{u'?\overrightarrow{x}=\overrightarrow{v},label=v_{label},\lambda x.Q} c' \qquad \langle \mathsf{service}\ u'(\overrightarrow{x}, label)\{P\}, \mu\rangle \in SS \qquad k\ \mathsf{fresh} \qquad B' = B :: \{(\lambda x.Q)^k\} \qquad CC' = CC'' \cup \{\langle\langle c', B', T\rangle^j\rangle^u\} \qquad SS' = SS \cup \{\langle\langle P, \mu\{\overrightarrow{x} \mapsto \overrightarrow{v}\}\rangle^k\rangle^j\}}{\langle SS, CC\rangle \rightsquigarrow^{u?\overrightarrow{x}=\overrightarrow{v},label=v_{label},\lambda x.Q} \langle SS', CC'\rangle}$$

This rule specifies how a client is able to make a HTTP Request to a server u' (that is, a server that could be different from its source server u) assigning the values $\overrightarrow{v}$ to the server parameters $\overrightarrow{x}$ and stating that its continuation will be in the form of the callback function $\lambda x.Q$. The callback function and the server configuration dedicated to that client are labelled with a fresh k that acts as a return

address, however the session is still labelled by j. The former is then moved to the callback queue in the client configuration, while the latter stores the parameter in its memory. Other than parameters $\overrightarrow{x}$ a client is able to express a value v_{label} for a specific variable, $label$ which specifies the desired interaction flow during the execution of the program.

RETSERVICE

$$\frac{SS = SS' \cup \{\langle\langle v_{\sim}, \mu\rangle^{k}\rangle^{j}\} \qquad CC = CC'' \cup \{\langle\langle c, \{(\lambda x.Q)^{k}\} :: B, T\rangle^{j}\rangle^{u}\} \qquad \mathsf{gencc}(v_{\sim}) = v \qquad CC' = CC'' \cup \{\langle\langle c, B, T :: \{(\lambda x.Q)v\}\rangle^{j}\rangle^{u}\}}{\langle SS, CC\rangle \rightsquigarrow \langle SS', CC'\rangle}$$

This rule specifies how a client may receive a response from a Server within a session j. This happens if there are both a client configuration and a server configuration labelled by j, if the server configuration contains a value and if the first callback function in the client configuration has the same return address k as the server configuration. The server returns value v resulting from $\mathsf{gencc}(v_{\sim})$ and applies the callback function to it, and adds the result to the thunks queue.

We present now the contextual rules that allow the internal moves of both clients and servers to take place within Webi configurations:

CLIENTSTEP

$$\frac{CC = CC'' \cup \{\langle\langle c, B, T\rangle^{j}\rangle^{u}\} \qquad c \rightsquigarrow_{C} c' \qquad CC' = CC'' \cup \{\langle\langle c', B, T\rangle^{j}\rangle^{u}\}}{\langle SS, CC\rangle \rightsquigarrow \langle SS, CC'\rangle}$$

When a client executes an internal computation, its configuration is updated accordingly.

SERVERSTEP1

$$\frac{SS = SS'' \cup \{s^{\,j}\} \qquad s \rightsquigarrow_S s' \qquad SS' = SS'' \cup \{s'^{\,j}\}}{\langle SS, CC\rangle \rightsquigarrow \langle SS', CC\rangle}$$

SERVERSTEP2

$$\frac{SS = SS'' \cup \{\langle s^{\,k}\rangle^j\} \qquad s \rightsquigarrow_S s' \qquad SS' = SS'' \cup \{\langle s'^{\,k}\rangle^j\}}{\langle SS, CC\rangle \rightsquigarrow \langle SS', CC\rangle}$$

Similarly, when a server makes an internal computation, its configuration is updated. Note that for the server we need two contextual rules.

RUN

$$\frac{CC = CC'' \cup \{\langle\langle\langle v', \mu\rangle, B, \{(\lambda x.Q)v\} :: T\rangle^j\rangle^u\} \qquad CC' = CC'' \cup \{\langle\langle\langle (Q\{x \mapsto v\}, \mu\rangle, B, T\rangle^j\rangle^u\}}{\langle SS, CC\rangle \rightsquigarrow \langle SS, CC'\rangle}$$

This rule executes one of the thunks, replacing an active client program that has reached a final configuration.

Chapter 7

WebiLog: Adding Login History to Webi

In the following we will focus on user-authenticated sessions established upon a successful login, in which cookies are used to authenticate users at the server side. In order to adapt Webi to model authenticated sessions we need to represent and detect login actions and record them into a lattice of security levels following the approach in (reference). A WebiLog configuration is now of this shape:

$$\langle SS, CC, L \rangle$$

where L is the Login History Lattice, whose elements are server names $u, u'...$, together with top and bottom elements $\top$ and $\bot$. At the beginning, we assume $L = \{\top, \bot\}$.

To record login actions, a special variable $login$ is introduced. Service declarations have now the form $\langle \mathsf{service}\ u(\overrightarrow{x}, login)\{P\}, \mu \rangle$ and correspondingly, client calls need to transmit both a vector of values $\overrightarrow{v}$ to be substituted for $\overrightarrow{x}$ and a value v_{login} to be substituted for $login$.

Given a configuration $\langle SS, CC, L \rangle$, a login on service u is performed in two

steps:

- A new client calls server u with a set of parameters $\overrightarrow{v}$ and a parameter $v_{login} \neq \mathsf{undefined}$.

  If $u \notin L$, this has to be a login request towards u, otherwise it should be a logged-in action on u.

- If server u replies with special value $cookie_{login(u)}$ then the authentication was successful, u is added to L and therefore, every subsequent call to u will have $v_{login} = cookie_{login(u)}$.

This behavior describes the use of cookies within a web browser.

7.1 WebiLog Semantics

We assume that a login action can only be performed when calling a server with a UrlRequest, therefore there are now three different alternatives for a client to start a session with a service $u(\overrightarrow{x}, login)\{P\}$, depending on the value assigned by the client to the parameter $login$ and the shape of the Login History L.

URLREQ

$$\frac{\langle \mathsf{service}\ u(\overrightarrow{x}, login)\{P\}, \mu\rangle \in SS \qquad j\ \mathsf{fresh} \qquad CC' = CC \cup \{\langle\langle \mathsf{boot}\rangle^j\rangle^u\} \qquad SS' = SS \cup \{\langle P, \mu\{\overrightarrow{x} \mapsto \overrightarrow{v}, login \mapsto \mathsf{undefined}\}\rangle^j\}}{\langle SS, CC, L\rangle \rightsquigarrow^{u?\overrightarrow{x}=\overrightarrow{v}, login=\mathsf{undefined}} \langle SS', CC', L\rangle}$$

A client makes its initial call to a server u which does not belong to L with login parameter equal to undefined, starting a non authenticated session.

URLLOGINREQ

$$\frac{\langle \mathsf{service}\ u(\overrightarrow{x}, login)\{P\}, \mu\rangle \in SS \qquad j\ \mathsf{fresh} \qquad u \notin L \qquad CC' = CC \cup \{\langle\langle \mathsf{boot}\rangle^j\rangle^u\} \qquad SS' = SS \cup \{\langle P, \mu\{\overrightarrow{x} \mapsto \overrightarrow{v}, login \mapsto v_{login}\}\rangle^j\}}{\langle SS, CC, L\rangle \rightsquigarrow^{u?\overrightarrow{x}=\overrightarrow{v}, login=v_{login}} \langle SS', CC', L\rangle}$$

A client issues an authentication request to a server u which does not belong to L, providing a value for the $login$ parameter. We are not adding u in L since we have to wait for the server response to judge the outcome of the login procedure.

URLREQ-LOGGEDIN

$$\frac{\langle \mathsf{service}\ u(\overrightarrow{x}, login)\{P\}, \mu\rangle \in SS \qquad j\ \mathsf{fresh} \qquad u \in L \qquad CC' = CC \cup \{\langle\langle \mathsf{boot}\rangle^j\rangle^u\} \qquad SS' = SS \cup \{\langle P, \mu\{\overrightarrow{x} \mapsto \overrightarrow{v}, login \mapsto cookie_{login(u)}\}\rangle^j\}}{\langle SS, CC, L\rangle \rightsquigarrow^{u?\overrightarrow{x}=\overrightarrow{v}, login=cookie_{login(u)}} \langle SS', CC', L\rangle}$$

A client issues an authentication request to the server u which belongs to L, meaning that the client has already authenticated to this server. The value assigned to the $login$ parameter of this request is therefore the login cookie for the server.

The response from a server to a Url request may alter the shape of the Login History, issuing a login cookie to the client.

CLIENTLOGIN

$$\frac{SS = SS' \cup \{\langle v_\sim, \mu\{login \mapsto cookie_{login(u)}\}\rangle^j\} \qquad CC = CC'' \cup \{\langle\langle \mathsf{boot}\rangle^j\rangle^u\} \qquad \mathsf{boot}(v_\sim) = Q \qquad CC' = CC'' \cup \{\langle\langle Q, \epsilon, \epsilon\rangle^j\rangle^u\} \qquad L' = L \oplus u}{\langle SS, CC, L\rangle \rightsquigarrow^{\mathsf{boot}} \langle SS', CC', L'\rangle}$$

The authentication was successful and the server responds with some code, the

Login History lattice L is updated with the name u and every subsequent request to it will carry the cookie.

CLIENTBOOT

$$\frac{SS = SS' \cup \{\langle v_\sim, \mu\rangle^j\} \qquad CC = CC'' \cup \{\langle\langle \mathsf{boot}\rangle^j\rangle^u\} \qquad \mathsf{boot}(v_\sim) = Q \qquad CC' = CC'' \cup \{\langle\langle Q, \epsilon, \epsilon\rangle^j\rangle^u\}}{\langle SS, CC, L\rangle \rightsquigarrow^{\mathsf{boot}} \langle SS', CC', L\rangle}$$

The server replies with code to the client, no login cookie is issued and an unauthenticated session is started.

A call from a client to a service named u is performed applying one of the following rules, depending on the structure of L . If the service name u does not belong to L, the following rule is applied:

CLIENTCALL

$$\frac{\langle \mathsf{service}\ u'(\overrightarrow{x}, login)\{P\}, \mu\rangle \in SS \qquad CC = CC'' \cup \{\langle\langle c, B, T\rangle^j\rangle^u\} \qquad c \rightsquigarrow_C^{u'?\overrightarrow{x}=\overrightarrow{v}, login=\mathsf{undefined}, \lambda x.Q} c' \qquad u \notin L \qquad k\ \mathsf{fresh} \qquad B' = B :: \{(\lambda x.Q)^k\} \qquad CC' = CC'' \cup \{\langle\langle c', B', T\rangle^j\rangle^u\} \qquad SS' = SS \cup \{\langle\langle P, \mu\{\overrightarrow{x} \mapsto \overrightarrow{v}\}\rangle^k\rangle^j\}}{\langle SS, CC, L\rangle \rightsquigarrow^{u'?\overrightarrow{x}=\overrightarrow{v}, login=\mathsf{undefined}, \lambda x.Q} \langle SS', CC', L\rangle}$$

If there is no current login on service u, the request is sent with the specified parameters without any cookie.

CLIENTCALL-LOGGEDIN

$$\frac{\langle \mathsf{service}\ u(\overrightarrow{x})\{P\}, \mu\rangle \in SS \qquad c \rightsquigarrow_C^{u'?\overrightarrow{x}=\overrightarrow{v}, login=cookie_{login(u)}, \lambda x.Q} c' \qquad u \in L \qquad k\ \mathsf{fresh} \qquad B' = B :: \{(\lambda x.Q)^k\} \qquad CC' = CC'' \cup \{\langle\langle c', B', T\rangle^j\rangle^u\} \qquad SS' = SS'' \cup \{\langle\langle P, \mu\{\overrightarrow{x} \mapsto \overrightarrow{v}, login \mapsto cookie_{login(u)}\}\rangle^k\rangle^j\}}{\langle SS, CC, L\rangle \rightsquigarrow^{u'?\overrightarrow{x}=\overrightarrow{v}, login=cookie_{login(u)}, \lambda x.Q} \langle SS', CC', L\rangle}$$

If a client calls a server on which an authentication request was successfully performed, the value of the login parameter is set to that of the login cookie issued by the server.

The remaining rules are nearly identical since they do not have to account for the Login History, and are left uncommented for brevity.

RETSERVICE

$$\frac{SS = SS' \cup \{\langle\langle v_\sim, \mu\rangle^k\rangle^j\} \qquad CC = CC'' \cup \{\langle\langle c, \{(\lambda x.Q)^k\} :: B, T\rangle^j\rangle^u\} \qquad \mathsf{gencc}(v_\sim) = v \qquad CC' = CC'' \cup \{\langle\langle c, B, T :: \{(\lambda x.Q)v\}\rangle^j\rangle^u\}}{\langle SS, CC, L\rangle \rightsquigarrow \langle SS', CC', L\rangle}$$

CLIENTSTEP

$$\frac{CC = CC'' \cup \{\langle\langle c, B, T\rangle^j\rangle^u\} \qquad c \rightsquigarrow_C c' \qquad CC' = CC'' \cup \{\langle\langle c', B, T\rangle^j\rangle^u\}}{\langle SS, CC, L\rangle \rightsquigarrow \langle SS, CC', L\rangle}$$

SERVERSTEP1

$$\frac{SS = SS'' \cup \{s^j\} \qquad s \rightsquigarrow_S s' \qquad SS' = SS'' \cup \{s'^j\}}{\langle SS, CC, L\rangle \rightsquigarrow \langle SS', CC, L\rangle}$$

SERVERSTEP2

$$\frac{SS = SS'' \cup \{\langle s^k\rangle^j\} \qquad s \rightsquigarrow_S s' \qquad SS' = SS'' \cup \{\langle s'^k\rangle^j\}}{\langle SS, CC, L\rangle \rightsquigarrow \langle SS', CC, L\rangle}$$

RUN

$$\frac{CC = CC'' \cup \{\langle\langle\langle v', \mu\rangle, B, \{(\lambda x.Q)v\} :: T\rangle^j\rangle^u\} \qquad CC' = CC'' \cup \{\langle\langle\langle(Q\{x \mapsto v\}, \mu\rangle, B, T\rangle^j\rangle^u\}}{\langle SS, CC, L\rangle \rightsquigarrow \langle SS, CC', L\rangle}$$

7.2 Examples

Suppose we have a bank service. In order to access the bank, one should first identify with its login interface $\langle \text{service } auth(login)\{P\}, \mu \rangle$ which will return him the code to interact with the bank and the login cookie if the authentification is correct. The code for P is:

```
login=check(login);
if (login = cookielogin) then {
  return ~(
    input = get input();
    if(input=transfer) then {
      call bank with
      [label=transfer, to=friend, amount=100] then do (
         \balance, label. branch (label) {
           done : return  balance;
           refused : return "error";
         })}
    else{
      if (input = balance) then{
        call bank with [label=balance] then do (
         \label, balance. branch (label) {
           done : return  balance;
           refused : return "error";
         })}
      else {
        call bank with [label=quit] then do (  \w.return w )
      }
     }
    )
```

```
  }
else
  return ~( skip )
```

We then have $\langle$service $bank(\overrightarrow{x}, login)\{P'\}, \mu\rangle$, where $\overrightarrow{x} = \{to, amount, label\}$, with the following P':

```
branch(label){ transfer:
                  if (login != undefined & valid(to, amount)) then
                      new-balance=update(amount);
                      return [label=done, new_balance]
                  else
                      return [label=refused]

         balance: if (cookielogin != undefined ) then
                      balance=balance();
                      return [label=done, balance]
                  else
                      return [label=refused, 0]

            quit: return true
       }
```

where we assume *balance()* and *update balance()* calculate and update the current balance of the user, *check()* authenticates the user and $islogincookie()$ checks the response of the authentication request.

We also have $\langle \mathsf{service}\ attacker(x)\{P''\}, \mu\rangle$ with the following P'':

```
return ~(
call bank with
    [label=transfer, to=attacker, amount=1000] then do (
         \balance, label. branch (label) {
           done : return  "Attack performed";
           refused : return "Attack blocked";
         })
      )
```

Function gencc:

$$\mathsf{gencc}(v) = v$$

Function boot():

$$\mathsf{boot}(v) = \{\!|v|\!\}(\mu)$$

7.2.1 Example 1: A secure WebiLog Session

In this example we describe how WebiLog can model the interaction between a server and a client. Suppose:

$$SS = \{\langle \textsf{service } auth(login)\{P\}, \mu\rangle, \langle \textsf{service } bank(\overrightarrow{x})\{P'\}, \mu\rangle\}$$

$$\langle SS \cup \{\langle \textsf{service } auth(login)\{P\}, \mu\rangle\}, \emptyset, L\rangle \rightsquigarrow^{auth?login=credentials}$$

$$\langle SS \cup \{\langle P, \mu\{login \mapsto credentials\}\rangle^j\}, \{\langle\langle \textsf{boot}\rangle^j\rangle^{auth}\}\}, L\rangle \rightsquigarrow^*$$

$$\langle SS \cup \{\langle \textsf{return } \tilde{}(\ldots), \mu\{login \mapsto cookie_{login(bank)}\}^j\},$$
$$\{\langle\langle \textsf{boot}\rangle^j\rangle^{auth}\}, L \oplus bank\rangle \rightsquigarrow^{\textsf{boot}}\rightsquigarrow^*$$

$$\langle SS \cup \langle \textsf{service } bank(\overrightarrow{x})\{P'\},$$
$$\{\langle\langle \textsf{call } bank \textsf{ with } \overrightarrow{x} := \{transfer, "friend", 100\} \textsf{ then do}$$
$$(\lambda\ label, balance.Q)\rangle, \emptyset, \emptyset\rangle^j\rangle^{auth}\}, L\rangle \rightsquigarrow^{bank?\overrightarrow{x}=\{transfer,"friend",100\},\lambda label,balance.Q}$$

$$\langle SS \cup \{\langle P', \mu\{\overrightarrow{x} \mapsto \{transfer, "friend", 100\}, login \mapsto cookie_{login(u)}\}\rangle^k\},$$
$$\{\langle\langle\emptyset, \{(\lambda\ label, balance.Q)^k\}, \emptyset\rangle^j\rangle^{auth}\}, L\rangle \rightsquigarrow^*$$

$$\langle SS \cup \{\langle \textsf{return } \{done, new - balance\}, \mu\rangle^j\},$$
$$\{\langle\langle\emptyset, \{(\lambda\ label, balance.Q)^k\}, \emptyset\rangle^j\rangle^{auth}\}, S, L\rangle \rightsquigarrow$$

$$\langle SS, \{\langle\langle\emptyset, \emptyset, \{(\lambda\ label, balance.Q)done\ new - balance\}\rangle^j\rangle^{bank}\}, L\rangle \rightsquigarrow *$$

$$\langle SS, \{\langle\langle \textsf{return } balance, \emptyset, \emptyset\rangle^j\rangle^{bank}\}, S, L\rangle$$

7.2.2 Example 2: An insecure WebiLog Session

In this example we show how Webi is able to represent Session Integrity violations such as CSRF attacks described before.

Suppose:

$$SS = \{\langle \textsf{service } auth(login)\{P\}, \mu\rangle,$$
$$\langle \textsf{service } bank(\overrightarrow{x})\{P'\}, \mu\rangle, \langle \textsf{service } attacker(x)\{P''\}, \mu\rangle\}$$

$$\langle SS \cup \{\langle \textsf{service } auth(login)\{P\}, \mu\rangle\}, \emptyset, L\rangle \rightsquigarrow^{auth?login=credentials}$$

$$\langle SS \cup \{\langle P, \mu\{login \mapsto credentials\}\rangle^j\}, CC \cup \{\langle\langle \textsf{boot}\rangle^j\rangle^{auth}\}\}, L\rangle \rightsquigarrow^*$$

$$\langle SS \cup \{\langle \textsf{return } \tilde{}(\ldots), \mu\{login \mapsto cookie_{login(bank)}\}^j\},$$
$$CC \cup \{\langle\langle \textsf{boot}\rangle^j\rangle^{auth}\}, L \oplus bank\rangle \rightsquigarrow^{\textsf{boot}}$$

$$\langle SS \cup \{\langle \textsf{service } attacker(x)\{P''\}, \mu\rangle\}, CC, L\rangle \rightsquigarrow^{attacker?x=\textsf{undefined}}$$

$$\langle SS \cup \{\langle P'', \mu\{x \mapsto \textsf{undefined}\}\rangle^j\}, CC \cup \{\langle\langle \textsf{boot}\rangle^j\rangle^{attacker}\}\}, L\rangle \rightsquigarrow$$

$$\langle SS \cup \{\langle \textsf{return } \tilde{}(\ldots), \mu\rangle^j\}, CC \cup \{\langle\langle \textsf{boot}\rangle^j\rangle^{attacker}\}, L\rangle \rightsquigarrow^{\textsf{boot}}$$

$$\langle SS \cup \langle \textsf{service } bank(\overrightarrow{x})\{P'\},$$
$$\{\langle\langle \textsf{call } bank \textsf{ with } \overrightarrow{x} := \{transfer, \text{"}attacker\text{"}, 1000\} \textsf{ then do}$$
$$(\lambda\ label, balance.Q)\rangle, \emptyset, \emptyset\rangle^j\rangle^{attacker}\},$$
$$L\rangle \rightsquigarrow^{bank?\overrightarrow{x}=\{transfer,\text{"}attacker\text{"},1000\},\lambda label,balance.Q}$$

$$\langle SS \cup \{\langle P', \mu\{\overrightarrow{x} \mapsto \{transfer, "attacker", 1000\}, login \mapsto cookie_{login(u)}\}\rangle\rangle^k\},$$
$$\{\langle\langle\emptyset, \{(\lambda\ label, balance.Q)^k\}, \emptyset\rangle^j\rangle^{auth}\}, L\rangle \rightsquigarrow^*$$

$$\langle SS \cup \{\langle \textsf{return}\ \{done, new - balance\}, \mu\rangle^j\},$$
$$\{\langle\langle\emptyset, \{(\lambda\ label, balance.Q)^k\}, \emptyset\rangle^j\rangle^{auth}\}, S, L\rangle \rightsquigarrow$$

$$\langle SS, \{\langle\langle\emptyset, \emptyset, \{(\lambda\ label, balance.Q) done\ new - balance\}\rangle^j\rangle^{bank}\}, L\rangle \rightsquigarrow *$$

$$\langle SS, \{\langle\langle \textsf{return}\ "Attack\ performed", \emptyset, \emptyset\rangle^j\rangle^{bank}\}, S, L\rangle$$

Chapter 8

Future work

8.1 Session types for WebiLog

We present a sketch of a Type System based on Session Types that could allow us to verify if a program of a server corresponds or not to the protocol that it must realize when it communicates with a client. For simplicity, we take into consideration only a subset of finite protocols protocols, leaving for future work the other cases, which include recursive protocols.

8.1.1 Types

Our types only capture the selection and branching constructs. This allows us for a simpler representation of the protocols and does not limit our approach since these constructs may be viewed as a generalisation of output and input.

The syntax for Global Types and Local Types is listed below and is very similar to the one given in sections 4.1, 4.2.

$$
\begin{array}{lrll}
\text{Global Types} & G & ::= & p \to q : \{l_i\langle\sigma_i\rangle : G_i\}_{i\in i} & \text{branching} \\
& & | & p \to_{login} q : (\overrightarrow{\sigma}); G' & \text{authentication} \\
& & | & end & \text{end}
\end{array}
$$

Introducing a Global type $p \to_{login} q : (\overrightarrow{\sigma}); G'$ for the authentication allows us to capture this particular type of interacton. This could also possibly allow us to enforce an Integrity property on the execution of session.

$$
\begin{array}{lrll}
\text{Local Types} & T & ::= & \oplus \langle q, \{l_i\langle\sigma_i\rangle : T_i\}_{i\in I}\rangle & \text{select} \\
& & | & \&(q, L, \sigma_1 \ldots \sigma_n)\{l_i : T_i\}_{i\in I} & \text{branch} \\
& & | & end &
\end{array}
$$

8.1.2 Type Checking

We suppose that every server is annotated with a Session type specifying its expected behaviour. This behaviour could be formalized, for example, in a communication protocol. Some services will offer several options in a branching construct, while others will perform an authentication procedure and return code to the client. Depending on the outcome of the procedure, different interactions will take place.

We consider only finite protocols which do not include while loops, in order to have a simpler environment and exclude recursive types. The typing rules include a standard typing environment Γ for variables, expressions and labels.

The rules are not properly formalized, and some required rules are missing, most importantly a rule for typing a main WebiLog configuration with a Global Type. A complete formulation of the Type System and of its Soundness is left as future work.

Server Program:

A Server may interact with the client in different ways. In the following rules we try to formalise how a Type Checker could be able to verify the adherence of these interaction with a specified protocol. The shape of type judgements for server and client programs is $\Gamma \vdash P \triangleright \Delta$ and $\Gamma \vdash Q \triangleright \Delta$where Δ is the sesson environment, associating a session type with each channels of the program. In these rules, $s[C]$ and $s[S]$ stand for the channels of the client and the server in session S.

RETURNCODE

$$\frac{\Gamma \vdash t \triangleright \Delta, s[C] : \ \&(S, \{login_i : T_i\}).T \qquad i \in \{0,1\}}{\Gamma \vdash \langle \mathsf{return} \ \ \tilde{}t, \mu\rangle \triangleright \Delta, s[S] : \ \oplus\langle C, \{login_i : T_i\}\rangle.end}$$

SERVERSELECTION

$$\frac{\Gamma \vdash e : \sigma \qquad label = l_i \quad \text{for some} \ \ i}{\Gamma \vdash \langle \mathsf{return} \ \ \{label, e\}, \mu\rangle \triangleright \Delta, s[p] : \oplus\langle q, \{l_i : T_i\}\rangle}$$

Rule RETURNCODE aims at ensuring that the code returned to the client is coherent with the result of the authentication of the client. Rule SERVERSELECTION aims at verifying that the server's code adheres to the behaviour of the branch that results from the evaluation of the client's call.

Client Program:

The only way in which a client is able to communicate with a server is through a client call.

CLIENTSELECTION

$$\frac{label \in L \qquad \Gamma \vdash \overrightarrow{e} : \sigma \qquad \Gamma \vdash (\lambda x.Q) \triangleright s[C] : T_i}{\Gamma \vdash \mathsf{call}\ u\ \mathsf{with}\ label, x := e\ \mathsf{then\ do}\ \lambda x.Q \triangleright \Delta, s[C] : \oplus\langle u, \{l_i\langle\sigma\rangle : T_i\}}$$

This rule aims at checking whether the client is supposed to execute this call to the server.

8.1.3 Examples

Recalling the Bank server example, given in 7.2, we can say that the identity provider $auth$ initiates a session of the following Global Protocol G:

$$bank \to c : \begin{cases} transfer\langle String, Int\rangle : & c \to bank : \begin{cases} done\langle Int\rangle : & end \\ refused\langle Bool\rangle : & end \end{cases} \\ balance : & c \to bank : \begin{cases} done\langle Int\rangle : & end \\ refused\langle Bool\rangle : & end \end{cases} \\ quit : & end \end{cases}$$

Projection of G onto its participants:

$$bank : \& \begin{cases} c, transfer\langle String, Int\rangle : & \oplus & \left\langle \begin{array}{rll} c, done\langle Int\rangle : & end \\ c, refused\langle Bool\rangle : & end \end{array} \right. \\ c, balance : & \oplus & \left\langle \begin{array}{rll} c, done\langle Int\rangle : & end \\ c, refused\langle Bool\rangle : & end \end{array} \right. \\ c, quit : & end \end{cases}$$

$$c : \oplus \begin{cases} bank, transfer\langle String, Int\rangle : & \& & \left\langle \begin{array}{rll} bank, done\langle Int\rangle : & end \\ bank, refused\langle Bool\rangle : & end \end{array} \right. \\ bank, balance : & \& & \left\langle \begin{array}{rll} bank, done\langle Int\rangle : & end \\ bank, refused\langle Bool\rangle : & end \end{array} \right. \\ bank, quit : & end \end{cases}$$

8.2 Session Integrity for WebiLog

During the internship we focused on Web Session Integrity, and since WebiLog provided an environment able to reproduce attacks on authenticated sessions, we propose to formalize this notion as a noninterference property along the lines of [18]. Specific adjustments regarding the formalization of WebiLog have been considered in order to achieve this goal, such as the incorporation of a set of streams S registering the inputs and outputs for all services considered in SS. A stream is defined by the following grammar:

$$S ::= \epsilon \mid s :: S$$

where s ranges over δ. In order to state a definition of noninterferece we would also need a suitable l-equality $\approx_l$ formalizing the LHD similarity between these streams and a labeling function which captures the correct level of integrity of client calls towards a specific service. A scribbled version of the LHD similarity $\approx_l$ and of the Session Integrity properties are shown below:

Definition (LHD-similarity)**.** *Given two streams S_1, S_2 we write $S_1 \approx_u S_2$ if we can build a proof tree for this statement using the following rules:*

ID-LOGIN

$$\frac{s = u?\overrightarrow{x} = \overrightarrow{v}, login = v_{login} \quad v_{login} \neq \mathsf{undefined} \qquad u \leq l \qquad L \oplus u \vdash S_1 \approx_l S_2}{L \vdash s \ :: \ S_1 \approx_l s \ :: \ S_2}$$

ID-SIM

$$\frac{s = u?\overrightarrow{x} = \overrightarrow{v}, login = v_{login} \quad v_{login} = \mathsf{undefined} \qquad lbl_L(s) \leq l \qquad L \vdash S_1 \approx_l S_2}{L \vdash s \ :: \ S_1 \approx_l s \ :: \ S_2}$$

ID-L

$$\frac{lbl_L(s) \not\leq l \qquad L \vdash S_1 \approx_l S_2}{L \vdash s \ :: \ S_1 \approx_l S_2}$$

ID-R

$$\frac{lbl_L(s) \not\leq l \qquad L \vdash S_1 \approx_l S_2}{L \vdash S_1 \approx_l s \ :: \ S_2}$$

ID-NIL

$$\frac{}{L \vdash \epsilon \approx_l \epsilon}$$

The key point in this approach is a correct definition of the labeling function lbl_L, which has to capture the correct level of integrity of the communication by altering its behaviour depending on the shape of the Login History L.

Definition (Session Integrity)**.** *A Server program* service $u(\overrightarrow{x})\{P\}$ *is noninterferent iff:* $\forall SS_1,\ SS_2$ *such that* service $u(\overrightarrow{x}, login)\{P\} \in SS_1 \wedge$ service $u(\overrightarrow{x}, login)\{P\} \in SS_2$ *and for every computation such that:*

$$\langle SS_1, \emptyset, \emptyset, \emptyset \rangle \rightsquigarrow^{I_1} \langle SS_1', CC_1, S_1, L_1 \rangle$$

$$\langle SS_2, \emptyset, \emptyset, \emptyset \rangle \rightsquigarrow^{I_2} \langle SS_2', CC_2, S_2, L_2 \rangle$$

we have that $\emptyset \vdash I_1 \approx_l I_2 \Rightarrow \emptyset \vdash S_1(u) \approx_l S_2(u)$.

In this definition we can see streams I_1, I_2 as sequences of decorated transitions occurring in WebiLog and thus recording only the inputs flowing towards

services, while $S_1(u), S_2(u)$ represent the stream of interleaved inputs and outputs of service u.

Conditions for ensuring this property were explored during my internship. The achievement of this goal is left for future work.

Conclusions

To summarize, the following topics were discussed in the previous chapters:

- We have presented, while discussing Secure Information Flow theory, the Security Lattice model, the notion of noninterference and a Security Type System, presenting an intuition of how it is able to achieve and guarantee integrity and confidentiality on the data that it manipulates.

- We have discussed Web Session Integrity in the context of authenticated sessions built on top of cookies, describing a class of attacks known as Session Hijacking and focusing on the case of CSRF Attacks. We have described approaches aiming at the enforcement of different Session Integrity notions, focusing in particular on the case of the formulation in terms of LHD noninterference. We introduced current issues and concerns related to privacy and security in the context of IoT systems.

- We have presented Session Types, a type theory focusing on the description and validation of communication protocols in the case of binary and multiparty sessions, describing the properties they can guarantee on the execution of the whole protocol by the means of local static checks.

- We have described the multitier paradigm, an approach towards web programming aiming at a simplificaton of the classical multitier architecture of

a web application by providing a single language to deploy tho whole system.We presented two languages adopting this paradigm: Hop and Links.

- We have presented Webi, a semantic framework to model interaction on the web. Clients and servers in Webi run programs written in a multitier language, inspired by Hop, of which we provided the semantics of both components.

- We have presented an extension of Webi, named WebiLog, in which it is possible to model authenticated sessions by means of a Login History Lattice.We have provided semantics that capture their behavior, including their vulnerabilities.

- We propose to further extend WebiLog with a Session Type based type system and a Login History Dependent noninterference property. The former could guarantee desirable safety properties, while the latter could ensure Session Integrity on authenticated sessions.

This thesis leaves many open research questions that will later be faced during my research activities. Webi extensions such as WebiLog could represent versatile tools in the process of formalization, analysis and verification of web environments. Another aim of this document is to convey a message of caution in the usage of today's available web applications and IoT systems, as they could pose significant risks towards our privacy, security and safety.

Bibliography

[1] ALRAWI, O., LEVER, C., ANTONAKAKIS, M., AND MONROSE, F. Sok: Security evaluation of home-based iot deployments. In *2019 2019 IEEE Symposium on Security and Privacy (SP)* (Los Alamitos, CA, USA, may 2019), IEEE Computer Society.

[2] BOUDOL, G., LUO, Z., REZK, T., AND SERRANO, M. Towards reasoning for web applications: an operational semantics for hop.

[3] BUGLIESI, M., CALZAVARA, S., FOCARDI, R., KHAN, W., AND TEMPESTA, M. Provably sound browser-based enforcement of web session integrity. vol. 2014, pp. 366–380.

[4] CALZAVARA, S., FOCARDI, R., SQUARCINA, M., AND TEMPESTA, M. Surviving the web: A journey into web session security. pp. 451–455.

[5] CALZAVARA, S., RABITTI, A., AND BUGLIESI, M. Sub-session hijacking on the web: Root causes and prevention. *Journal of Computer Security 27* (10 2018), 1–25.

[6] CALZAVARA, S., RABITTI, A., RAGAZZO, A., AND BUGLIESI, M. *Testing for Integrity Flaws in Web Sessions*. 09 2019, pp. 606–624.

[7] CELIK, Z. B., MCDANIEL, P., AND TAN, G. Soteria: Automated iot safety and security analysis. In *2018 USENIX Annual Technical Confer-*

ence (USENIX ATC 18) (Boston, MA, July 2018), USENIX Association, pp. 147–158.

[8] CHECKOWAY, S., MCCOY, D., KANTOR, B., ANDERSON, D., SHACHAM, H., SAVAGE, S., KOSCHER, K., CZESKIS, A., ROESNER, F., AND KOHNO, T. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX Conference on Security* (USA, 2011), SEC'11, USENIX Association, p. 6.

[9] CHU, G., APTHORPE, N., AND FEAMSTER, N. Security and privacy analyses of internet of things children's toys. *IEEE Internet of Things Journal 6*, 1 (Feb 2019), 978–985.

[10] COOPER, E., LINDLEY, S., WADLER, P., AND YALLOP, J. Links: Web programming without tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects* (Berlin, Heidelberg, 2006), FMCO'06, Springer-Verlag, pp. 266–296.

[11] DENNING, D. E. A lattice model of secure information flow. *Commun. ACM 19*, 5 (May 1976), 236–243.

[12] FOWLER, S., LINDLEY, S., MORRIS, J. G., AND DECOVA, S. Exceptional asynchronous session types: Session types without tiers. *Proc. ACM Program. Lang. 3*, POPL (Jan. 2019).

[13] GOGUEN, J. A., AND MESEGUER, J. Security policies and security models. In *IEEE Symposium on Security and Privacy* (1982), pp. 11–20.

[14] HEDIN, D., AND SABELFELD, A. Information-flow security for a core of javascript. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium* (USA, 2012), CSF '12, IEEE Computer Society, pp. 3–18.

[15] HONDA, K., VASCONCELOS, V. T., AND KUBO, M. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *Proc. ESOP'98* (1998), vol. 1381 of *LNCS*, Springer, pp. 22–138.

[16] HONDA, K., YOSHIDA, N., AND CARBONE, M. Multiparty Asynchronous Session Types. In *Proc. POPL'08* (2008), ACM Press, pp. 273–284.

[17] JIA, Y. J., CHEN, Q. A., WANG, S., RAHMATI, A., FERNANDES, E., MAO, Z. M., AND PRAKASH, A. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *21st Network and Distributed Security Symposium* (Feb 2017).

[18] KHAN, W., CALZAVARA, S., BUGLIESI, M., GROEF, W., AND PIESSENS, F. Client side web session integrity as a non-interference property. vol. 8880, pp. 89–108.

[19] LINDLEY, S., AND MORRIS, J. G. A semantics for propositions as sessions. In *Programming Languages and Systems* (Berlin, Heidelberg, 2015), J. Vitek, Ed., Springer Berlin Heidelberg, pp. 560–584.

[20] LINDLEY, S., AND MORRIS, J. G. Lightweight functional session types.

[21] MIRAZ, D., ALI, M., EXCELL, P., AND PICKING, R. A review on internet of things (iot), internet of everything (ioe) and internet of nano things (iont). pp. 219–224.

[22] MOSTROUS, D., AND VASCONCELOS, V. T. Affine sessions. In *Coordination Models and Languages* (Berlin, Heidelberg, 2014), E. Kühn and R. Pugliese, Eds., Springer Berlin Heidelberg, pp. 115–130.

[23] PEDERSEN, M. V., AND ASKAROV, A. From trash to treasure: Timing-sensitive garbage collection. *2017 IEEE Symposium on Security and Privacy (SP)* (2017), 693–709.

[24] SERRANO, M., GALLESIO, E., AND LOITSCH, F. Hop: a language for programming the web 2.0. pp. 975–985.

[25] VOLPANO, D., IRVINE, C., AND SMITH, G. A sound type system for secure flow analysis. *J. Comput. Secur. 4*, 2-3 (Jan. 1996), 167–187.

[26] ZHANG, N., MI, X., FENG, X., WANG, X., TIAN, Y., AND QIAN, F. Dangerous skills: Understanding and mitigating security risks of voice-controlled third-party functions on virtual personal assistant systems. In *2019 IEEE Symposium on Security and Privacy (SP)* (May 2019), pp. 1381–1396.

[27] ZHANG, W., MENG, Y., LIU, Y., ZHANG, X., ZHANG, Y., AND ZHU, H. Homonit: Monitoring smart home apps from encrypted traffic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2018), CCS '18, Association for Computing Machinery, pp. 1074–1088.