Université Paris Diderot (Paris 7)
École doctorale de Sciences Mathématiques de Paris Centre

Thèse
pour obtenir le grade de
Docteur de l'Université Paris Diderot
Spécialité : **Informatique**

Présentée par
Nataliya Guts

Directeurs : Jean-Jacques Lévy et Francesco Zappa Nardelli

Auditabilité pour les protocoles de sécurité

Auditability for security protocols

Soutenue le 11 janvier 2011
devant le jury composé de :

M. Gilles Barthe	rapporteur
M. Roberto Di Cosmo	président du jury
M. Cédric Fournet	examinateur
M. Carl Gunter	examinateur
M. Robert Harper	rapporteur
M. Francesco Zappa Nardelli	directeur

Résumé : Les protocoles de sécurité enregistrent souvent des données disponibles lors de leurs exécutions dans un journal pour une éventuelle analyse *a posteriori*, aussi appelée audit. En pratique, les procédures d'audit restent souvent informelles, et le choix du contenu des journaux est laissé au bon sens du programmeur. Cette thèse a pour but la formalisation et la vérification des propriétés attendues des journaux d'audit.

D'abord, nous nous intéressons à l'utilisation des journaux par les protocoles de sécurité dits *optimistes* qui, contrairement aux protocoles classiques, reposent sur le contenu des journaux pour remettre certaines vérifications de sécurité à la fin de leur exécution. Nous faisons une étude formelle de deux schémas optimistes : la mise en gage de valeur et le porte-monnaie électronique. En appliquant les techniques issues des langages de processus, nous montrons que les informations enregistrées par leurs implémentations suffisent pour détecter toute tentative de tricherie de participants.

Ensuite nous définissons l'*auditabilité* comme la capacité d'un protocole de collecter assez de preuves pour convaincre une procédure d'audit préétablie (juge). Nous proposons une méthode basée sur les types avec des raffinements logiques, pour vérifier l'auditabilité, et nous l'implémentons dans une extension d'un typeur existant. Nous montrons que la vérification de l'auditabilité se réduit à une vérificationpar le typage. Nous implémentons également un support logique des pré- et post-conditions génériques pour améliorer le typage modulaire des fonctions d'ordre supérieur.

Mots clés : Journal d'audit, propriétés de sécurité, protocole cryptographique, système de types.

Abstract: Security protocols often log some data available at runtime for an eventual *a posteriori* analysis, called audit. In practice, audit procedures remain informal, and the choice of log contents is left to the programmer's common sense. The goal of this dissertation is to formalize and verify the properties expected from audit logs.

First we consider the use of logs in so called *optimistic* security protocols which, as opposed to classic security protocols, rely on the logs to postpone certain security checks until the end of execution. We formally study two optimistic schemes: value commitment scheme and offline e-cash; using process languages techniques, we prove that the information logged by their implementations suffices to detect the cheat of participants, if any.

Then we define *auditability* as the ability of a protocol to collect enough evidence to convince an audit procedure (judge). We propose a method based on types with logical refinements to verify auditability, and implement it as an extension to an existing typechecker. We show that verifying auditability boils down to typechecking the protocol implementation. We also implement logical support for generic pre- and post-conditions to enhance modular typechecking of higher-order functions.

Keywords: Audit logs, security properties, cryptographic protocols, type system.

Ces travaux ont été menés au Centre de Recherche Commun INRIA - Microsoft Research – Parc Orsay Université 28, rue Jean Rostand 91893 Orsay.

Acknowledgements

I'm deeply grateful to Francesco Zappa Nardelli, Cédric Fournet, and Jean-Jacques Lévy for their guidance, coaching, and friendship. In particular, I owe a lot to Cédric who has been a great, though unofficial, supervisor.

For the last part of the thesis I had the pleasure of working with Karthikeyan Bhargavan. I'm also grateful to Karthik and Cédric for offering me the opportunity of internship at Microsoft Research in Cambridge. I also enjoyed working with Pedro Adão during his visits to the Microsoft Research - INRIA Joint Centre. I'd like to thank all the colleagues, and Martine particularly for her fabulous assistance.

I'd like to thank the members of my jury, and to Gilles Barthe and Robert Harper in particular for accepting to review this manuscript.

Many thanks to my parents and friends. Thanks to Gabriel for being cute. The biggest thanks go to Alexis ♡. Without him I would have neither got the idea to start a PhD nor had the courage to finish it. And I would have definitely lost more battles against LaTeX, which is nevertheless a brilliant tool.

Contents

Chapter 1

A formal approach to audit logs

Most applications write audit logs. We are used to system logs: large text files where the operating system records who-what-where-when about all the notable events that occurred during each session. When something bad happens, the administrator just has to look at the logs to identify the problem and understand its causes. In practice, the amount of data logged is huge, most of the stored data is irrelevant for a given problem, and the success of the analysis of the logs is determined by the administrator's forensic skills.

System logs are an example of audit logs with weak application properties; they only outline the sequencing of the system events, provided that the superuser who generated them is trusted. Audit logs can also be used to enforce stronger security properties: in this case more complex cryptographically protected data is stored and can be used as out-of-context proofs. For instance, servers of web applications may log all authenticated connections and the history of every transaction with the clients. Server and clients may mistrust each other, and the logs should contain reliable cryptographic evidence for each party.

Audit logs are usually used as an extra protection mechanism, along with other classic runtime mechanisms to enforce the security goals. For some applications though, relying on logs is the most efficient, or even the only available protection mechanism. Some runtime security checks can then be discarded and the data relevant for these checks can be securely recorded in a log: the security checks can be performed *a posteriori*. We call this approach *optimistic*. This is especially helpful in domains where cheating and security attacks are rare: the guarantee that the attack will be discovered and punished is reasonably strong and eliminates the cost of continuously checking the good behaviour of all parties. The same principle applies as for the penal laws: crimes are only investigated when discovered. Because of the punishment, incentives for malicious behaviour fade away. The optimistic approach obviously does not apply to critical applications (e.g. military applications). But they apply particularly well when quick decisions benefit the application while introducing little additional risk. Automatic business applications earn money by reacting very quickly to uncertain information about the stock market, in cases where the risk of money loss is limited compared to the eventual money gain. For medical applications, in case of an emergency, the usual access control mechanisms should be bypassed to access the health record of the patient immediately; unauthorized access is a minor risk when compared to the threat to the patient's health, and must be recorded anyway [Becker and Sewell, 2004, Barth et al., 2007]. In online network gaming applications responsiveness is crucial for the game; secure logging may be used to double-check the consistency of the adversarial moves *a posteriori* to avoid a slowdown during the game [Jha et al., 2007].

In this work Audit logs are recognized as a valid security enforcement mechanism even if their analysis remains hard. A large body of studies covers forensic analysis and intrusion detection techniques (see the RAID or DFRWS conference series, for example). The development of these analyses is vital, as they can be parameterized by patterns of attacks or properties and thus can

exploit at best the available logged data for multiple security goals. However, some applications, including but not limited to optimistic protocols, rely on audit logs only to enable checking a given, pre-established security goal. We advocate that in these cases a formal approach can guide the programmer to select the data that must be logged, and can guarantee that the logged evidence will always suffice to check the given security goal. This work is thus focused on the following problem:

for a given application and security goal, which data should be logged?

Apart from general recommendations such as "an audit trail should include sufficient information to establish what events occurred and who (or what) caused them" [NIST, 1996, ISO/IEC, 2004] and efficient implementation techniques, we are not aware of any formal studies that characterize and verify the security properties achieved by protocols relying on logs.

Contributions In this work we explore two aspects of the problem sketched above. First we study how process language techniques can be used to prove that a given protocol logs enough information to ensure that cheating can be detected and that cheaters can be identified *a posteriori*. Second, we focus on programs written in a dialect of ML and we propose a type discipline based on dependent types so that verifying whether enough data is logged boils down to typechecking. In more details, our contributions are as follows:

Optimistic security We investigate the properties of a sample optimistic scheme, called value commitment. Value commitment enables a principal to commit to a value without revealing it, while being able to prove to the other principals that he really committed to a particular value. The value commitment scheme is a building block of many optimistic protocols, such as sealed bids auctions, and protocols using the cut-and-choose schema [Chaum et al., 2004, Chaum, 2004, Neff, 2001]. We design an extension of the applied pi calculus [Abadi and Fournet, 2001] which supports *committable cells*: these are write-once cells equipped with a blinded receipt of commitment. This receipt can be shown to other principals to prove the commitment and does not reveal the committed value. The semantics of committable cells builds in the expected properties of binding (the principal did commit to a value, the principal cannot change the value he committed to) and hiding (the value is not revealed by the receipt). Value commitment is typically implemented using cryptographic hash functions. We compile committable cells to standard cryptographic constructs of the applied pi calculusand store them in a log. We establish a full abstraction result between the two semantics, thus showing that the expected security properties are enforced by our use of cryptography and logs. More than that, this shows that the implementation guarantees that any cheating (e.g. multiple commitments to the same cell) will be detected, and the logged evidence will allow to blame the guilty party.

We call this property *optimistic security*: either the security goals are achieved or there is reliable evidence such that misbehaving principals can be blamed. In general, principled use of logs carefully generated for a particular purpose – and so designed to be given as input to a particular decision (audit) procedure – makes the application *auditable* for this procedure, and for the corresponding application level property.

We apply a similar methodology to study an existing implementation of an offline e-cash protocol [Camenisch et al., 2005]. This implementation is cryptographically more sophisticated than the one we designed for value commitment. This time we extract an idealized model that by construction enjoys the expected security properties. Again, we establish a full abstraction result between the idealized semantics and the symbolic protocol implementation, proving that it satisfies optimistic security.

These two test cases show that the approach based on process languages can then be used both to design a protocol implementation (value commitment) and to formalize the protocol

abstract behaviour (ecash). Also they show that optimistic security is a useful and precise property enforced by optimistic protocols.

Verification of auditability Although the pi calculus is a convenient and useful tool for reasoning about protocols, real life protocol implementations are written in full-fledged programming languages. We aim to define and verify the properties related to audit logs for realistic protocol implementations. Operating at the level of the protocol source code ensures that both design and implementation flaws will be caught, and also facilitates the adoption of verification tools by programmers. We choose to study protocol implementations written in F7 [Bengtson et al., 2008b, Bhargavan et al., 2010a], a dialect of ML with logical refinements.

First, we isolate and propose a formal definition of *auditability*, which we believe is the most useful and the most general security property provided by audit logs. Second, we propose a method for automated verification of auditability using the F7 types.

General definition of auditability Suppose that we are given a protocol implementation with some specific type annotations (for instance when accepting a message or allocating a key, the corresponding event is recorded as a logical assumption and as a refinement of the types of all related values).

We propose to annotate the code with *audit goals*: the programmer may claim at some points in the program (that we call *audit points*) to have collected evidence that, if logged and analysed *a posteriori* allows to "pass" the audit – that is, to convince the auditor (or *judge*) of this goal. The role of the judge is crucial: intuitively, his decision procedure specifies the *rules* of audit; all parties concerned by audit must know and accept the judge in advance, in particular to acknowledge his trust assumptions. We define *auditability* of a program for a property as the ability of this program to convince the judge specialized for this property at every audit point using the evidence collected at that point. For instance, the implementation of the value commitment scheme is auditable for the property "Whenever a principal is blamed, he attempted a multiple commitment of a cell".

Checking auditability We rely on F7, an SMT-based typechecker developed for the modular, automatic verification of cryptographic protocols. The main idea for checking auditability is to typecheck the source code of the judge and that of the program which embeds the audit points and verify that the type of the judge function matches the type of the evidence at every audit point. We illustrate this typing discipline by statically checking auditability of our examples, including a multi-party protocol.

Flexible pre- and post-conditions F7 allows to verify protocols that use recursive data structures (for instance, lists in our multi-party protocol example). However this entails replicating higher-order library functions and annotating each instance with its own logical pre- and post- conditions. Instead, we equip higher-order functions with precise, yet reusable types that can refer to the pre- and post-conditions of their functional arguments, using generic logical predicates. We implement our method by extending the F7 typechecker with automated support for these predicates. Experimentally this approach allows to reduce the amount of type annotations for verifying auditability of our examples. As an additional benefit, it allows to express the typing requirement that the post-condition of the audit procedure must match the pre-condition at all audit points.

Structure of the document In Chapter 2 we recall the research work we are building on and we define our main notations. In Chapter 3 we formally study the use of audit logs in the implementations of optimistic protocols: we design and study a formal implementation of value commitment, then we apply the same technique to study an existing implementation of electronic cash. In Chapter 4 we propose a general definition of auditability and illustrate its use for our case studies. In Chapter 5 we present a method for automatically checking auditability

of program code annotated with logical formulas using the F7 typechecker. In Chapter 6 we extend the F7 typechecker with generic pre- and post-conditions and use this extension to make the annotations more compact and modular.

Related works are discussed through the document. Conclusions and possible future work are discussed at the end of Chapters 3 and 6. The source code for all our protocols as well as the instructions for typechecking it, are available online [cod].

Provenance of the material This dissertation is partially based on published material.

The case study on optimistic implementation of value commitment has been published in the proceedings of the 17th European Symposium on Programming [Fournet et al., 2008] and presented in April 2008 in Budapest, Hungary. A very preliminary formalization of e-cash has been presented in the 4th Workshop on Formal and Computational Cryptography [Adão et al., 2008] in June 2008 in Pittsburgh, USA. An earlier version of the work on general definition of auditability has been published in the proceedings of the 14th European Symposium on Research in Computer Security [Guts et al., 2009] and presented in September 2009 in Saint-Malo, France; this dissertation uses a more recent version of the F7 verification method. The work on the use of pre- and post-conditions for security typechecking will appear in the proceedings of the 8th Asian Symposium on Programming Languages and Systems [Bhargavan et al., 2010b].

Chapter 2

Preliminaries

In this chapter we introduce the notations we use in this thesis (Section 2.1), we recall and adapt some research works we build on (Sections 2.3 and 2.4), and, perhaps more interestingly, we describe once for all several protocols we use all along this thesis (Section 2.2).

2.1 Security protocols: notation and goals

A protocol is a set of rules on message exchange aiming to achieve security goals between two or more partners.

To describe protocols we adopt notations common in the literature. Protocol participants (also called principals, agents, parties, etc.) are denoted with capitals $A, B, \ldots, X$. The transmission of message m from entity X to participant Y is denoted $X \rightarrow Y : m$. We use two sets of notations for messages. To specify a particular implementation of a cryptographic protocol, we use the common "concrete" syntax:

$m ::=$	Messages
N	nonce (fresh name)
$h(m)$	a collision resistant one-way hash function
$\{\ \}_k$	encryption with key k
$m_1 \vert m_2$	concatenation
$k ::=$	Keys
k_{XY}	symmetric key shared between X and Y
pk_X	public key for signature verification and encryption for X
sk_X	private key of entity X, used for signing and decryption

In this thesis we often assume the existence of a trusted public key infrastructure (PKI) which binds public keys to the identities of the participants.

In addition, to abstract away the concrete cryptographic implementation of a message and highlight its security properties, we use an "abstract" syntax:

$m ::=$	
N	nonce (fresh name)
$m_1 \vert m_2$	concatenation
$authentic(m, X)$	message m, authenticated by principal X
$secret(m, X)$	message m, only readable by principal X
$blinded(m)$	commitment to later reveal message m

Security properties Below we informally discuss different security goals of protocols.

Secrecy (also called confidentiality) of data is about restricting read access to this data. For example, the passphrase typed by the user to access some web server should only be received by the server, and in particular remain unknown to the attacker.

Anonymity guarantees that the identity of a principal is not revealed by the messages exchanged during the protocol. Privacy is a related property: for example, one may not want his browsing history on Youtube to be linked to his identity to preserve his privacy.

Integrity of data is about restricting write access to this data: only certain principals can modify the data. For example, when downloading a file from a distant server, you may want to be sure that what you receive is the correct content of the file, and that no transmission errors occurred.

Authentication means that the data is known to be related (issued, received) from a particular principal. For instance, when logging to your webmail, you type your password to authenticate yourself and your actions.

Other properties may be targeted as protocol goals: fairness (all partners enjoy the same guarantees), non-repudiation (inability to deny an action), verifiability (the ability of one or more principals to verify the outcome of the protocol) and others.

This thesis studies another security property called *auditability*. We will introduce and define it in Chapter 4.

Example 2.1.1. *Suppose that student Alice sends her homework essai.txt to professor Bob for grading. Alice trusts the professor but has no illusions about the public channel used for submissions: its contents can be read and modified by anyone from the university campus.*

*To protect her work from being stolen, she encrypts it with the public key of the professor, so that he is the only one to be able to decrypt it using his private key (*secrecy*). To convince the professor that she is the sender of the homework (*authentication*) and that the homework has not been corrupted (*integrity*), she encloses the digital signature of the message using her private key.*

Using our concrete notation, we write the homework mini-protocol as

$$Alice \;\rightarrow\; Bob : \; \{essai.txt\}_{pk_{Bob}} \,|\, \{\{essai.txt\}_{pk_{Bob}}\}_{sk_{Alice}}$$

In its abstract form, the protocol is written as

$$Alice \;\rightarrow\; Bob : \; authentic(secret(essai.txt, Bob), Alice)$$

Attacks The goals of the protocols must be achieved even when the protocol runs in the presence of hostile parties, attackers. We consider the *active attacker* [Dolev and Yao, 1983] who can eavesdrop, modify, or substitute, any message exchanged on the network. The attacker can also initiate a protocol session and actively participate in a session, using his own credentials or the credentials of the parties he corrupted.

Some kinds of the attacks have been extensively studied and described in the literature, for example replay attacks or attacks by reflexion. Abadi and Needham [1996] describe the practical solutions which offer a shield against such attacks. Our sample protocols may not be protected against all known attacks; instead, we focus on implementing the protections that support our property of interest, namely auditability.

2.2 The protocols used in the following chapters

In this section we informally describe several protocols that will serve as examples in this thesis. We are going to experiment with their implementations, and reason about their formal properties in the following chapters.

2.2.1 Simple message authentication

Consider a simple protocol where a client Alice sends an authenticated mail to a server Bob.

$$Alice \rightarrow Bob: \ authentic(message, Alice)$$

To prove her identity, Alice produces some evidence, symbolically represented as the term $authentic(message, Alice)$, that Bob can check. We implement this evidence in several ways in the following chapters. For example, in Section 4 *authentic* is implemented by including the digital signature of the message with Alice's private key: Bob can verify it using Alice's public key.

Intuitively, this protocol guarantees integrity and authenticity of the message. Bob accepts the message if and only if the verification of the evidence against the message succeeds. He relies on the unforgeability of the received evidence: only Alice could have issued it, so she indeed intended to send the message, and the message was not modified by the environment.

2.2.2 Rock-Paper-Scissors

A server Charlie organizes a session of Rock-Paper-Scissors between Alice and Bob. Charlie is trusted by Alice and Bob in the following respects: Charlie does not help any of the players to choose a move, and Charlie does not to tamper with the received moves. The function *winning* returns Rock for Scissors, Scissors for Paper, and Paper for Rock, as expected.

$$\begin{aligned} Alice &\rightarrow Charlie : authentic(mA, Alice) \\ Bob &\rightarrow Charlie : authentic(mB, Bob) \\ Charlie &\rightarrow Alice, Bob : winning(mA, mB) \mid authentic(mA, Alice) \mid authentic(mB, Bob) \end{aligned}$$

Without the trust assumption, Charlie could obviously help Bob by forwarding Alice's move to him so that he can choose the winning move. Note that no secrecy is provided, so the adversary can eavesdrop Alice's message and still help Bob.

The main goals of this protocol are integrity and authentication of the players' moves.

2.2.3 Multi-party game

The game is run by a server $Server$ between n players $Player_1, \ldots Player_n$. Unlike in the previous example, the server is untrusted and may collude with other dishonest players. The game is played in one turn, with all players revealing their moves simultaneously. For simplicity, we assume that the game is symmetric between all players. The protocol participants are willing to cooperate but with minimal trust assumptions between them; however, it is deemed sufficient to detect any dishonest principal at the end of the game. They also want to reveal as little information as possible; in particular they do not reveal their moves until everyone has played (as e.g. in the Lockstep protocol [Baughman and Levine, 2001]).

At the end of the game, depending on the moves for all players, one player wins, and expects to be recognized as the winner. The game may be instantiated to Rock-Paper-Scissors, online auctions (as in our protocol description), leader elections, and similar partial-information protocols [Shamir et al., 1981, Castellà-Roca et al., 2003, Chaum et al., 2004].

The protocol has three exchange rounds between the server and each player:

(1) the server sets up the game (in particular generating a fresh game identifier id), distributes the details to the players, and collects their sealed moves;

(2) the server distributes all the players' sealed moves and collects their actual moves;

(3) the server distributes the result of the game (the function *winning* returns the winning move for a given list of moves according to the game rules); as a variant of the protocol, the server can also distribute all the players' moves.

$$
\begin{aligned}
&Server \;\rightarrow\; Player_i : id\\
&Player_i \;\rightarrow\; Server : authentic(blind(id \,|\, m_i), Player_i)\\
&Server \;\rightarrow\; Player_i : authentic((id, blind(id \,|\, m_i)_{i\in 1..n}), Server)\\
&Player_i \;\rightarrow\; Server : m_i\\
&Server \;\rightarrow\; Player_i : winning(m_{1..n})
\end{aligned}
$$

The goals of this protocol include integrity and authentication of the players' moves, but also integrity and authentication of the server's commitment at the end of the first round. In addition, the secrecy of the players' moves is maintained until the end of the second round when they intentionally disclose their moves. Note that an instance of the game with $n = 2$ implements the Rock-Paper-Scissors protocol providing additional security properties.

2.3 Applied pi calculus

In the first part of this thesis we formalize protocols using the applied pi calculus. We refer to Abadi and Fournet [2001] for a general presentation of its semantics. Below we briefly recall its syntax and semantics.

The applied pi calculus is a process language parameterized by an equational theory on terms, which provides flexible support for modelling symbolic cryptographic primitives and data structures. It is based on Milner's pi calculus [Milner, 1999].

$M,\ V$	::=	$P,\ Q,\ R$	::=	$A,\ S$	::=
	u		0		$p[P]$
	$M + M'$		$P_1 \,|\, P_2$		$A_1 \,|\, A_2$
	$func(\tilde{M})$		$\mathbf{new}* \, c \,.\, P$		$\nu \, u \,.\, A$
			$u?(M).P$		$\{M/x\}$
			$u!\langle M\rangle.P$		
			$\mathsf{if}\ M = M'\ \mathsf{then}\ P\ \mathsf{else}\ P'$		
			$\mathsf{repl}\ P$		

Figure 2.1: Terms, processes, and systems in applied pi calculus

Syntax The grammar for terms (M, V), processes (P), and systems (A) is given in Figure 2.1.

Terms contain names, variables (denoted $x, y, \ldots$) and function applications respecting the signature. Among names we distinguish communications channels ranged over by c, nonces denoted $l, s, n, \ldots$, and principals denoted $p, a, e, Alice, Bob$. The metavariable u ranges over names and variables.

The set of terms is parameterized with a *signature*: a set of function symbols with arities, for instance $+/2$,$+_1/1$. A signature is equipped with an equivalence relation for terms, called *equational theory*, often generated by a set of rewrite rules. We assume that functions include at least a pairing function, denoted $+$, with associated projections $+_1, +_2$ and equations $+_i(x_1+$

$x_2) = x_i$ for $i = 1, 2$. Our examples will introduce additional data structures and we also use integer constants.

We rely on standard symbolic cryptographic primitives:

- public-key signature and encryption mechanisms with constructors $\mathsf{pk}/1$, $\mathsf{sk}/1$, $\mathsf{verify}/3$, $\mathsf{sign}/2$,$\mathsf{ok}/0$, $\mathsf{dec}/2$, $\mathsf{enc}/2$. The functions $\mathsf{sk}(m)$ and $\mathsf{pk}(m)$ generate a pair of secret/public keys from a nonce m. The function $\mathsf{verify}(m\,,\,s\,,\,k)$ models verification of a signature s against a message m using the public key k. The function $sign(m, k)$ generates a signature for a message m using the secret key k. The constant ok allows to test the result of signature verification. The functions $dec(m, k)$ (resp. $enc(m, k)$ model decryption and encryption of a message m using with the private (resp. public) key k. These constructors satisfy the equations:
$$\mathsf{verify}(v\,,\,\mathsf{sign}(v\,,\,\mathsf{sk}(m))\,,\,\mathsf{pk}(m)) = \mathsf{ok}$$
$$\mathsf{dec}(\mathsf{enc}(v\,,\,\mathsf{pk}(m))\,,\,\mathsf{sk}(m)) = v$$
- hash function, denoted $\mathbf{h}/1$, with no equations. Not providing an equational axiom is meaningful: the hash function is assumed to be one-way only. The term $\mathbf{h}(M)$ can be constructed from the term M but the term M cannot be extracted from its hash.

A concurrent system is modelled as composition of processes locally run by principals. Plain processes, usually called simply *processes* include parallel composition, replication, generating a fresh name, output of a term (and not just a name as in pi calculus), input on channel, conditional construct where the condition is evaluated according to the equational theory. Contrarily to the original applied pi calculus, each process P runs under the control of a principal p, denoted $p[P]$.

Extended processes, also called *systems*, include processes, parallel composition of systems and name and variable restrictions and *active substitutions*: $\{^M/_x\}$ replaces the variable x with the term M in all parallel processes as far as the restriction of x allows. So $\nu x.(\{^M/_x\}|P)$ corresponds exactly to $let\, x = M\, in\, P$. By replacing all processes $p[P]$ in an extended process A with the null process 0, we obtain its *frame* $\phi(A)$ that approximates the static information exposed by A to the environment through the skeleton of its restrictions and active substitutions. For example, the frame $\nu\, s\,.\,\nu\, y\,.\,\{\,\mathbf{h}(s)\,/\,x\,\}\ \mid\ \{\,(\,s\ +\ s\,)\,/\,y\,\}$ exports the variable x while the variable y is under restriction: the environment thus knows the hash of some secret s, but not its concatenation with itself.

We denote $fv(A)$, $fn(A)$, $bv(A)$ and $bn(A)$ the sets of free and bound variables and names of an extended process, respectively. We abbreviate successive restrictions $\nu u_1, \ldots, \nu u_n.A$ as $\nu \tilde{u}.A$. Tuples $M_1, \ldots, M_n$ are abbreviated as $\tilde{M}$.

Alice's and Bob's roles from Example 2.1.1 can be modelled in applied pi calculus as shown below, we assume that c is the public channel they use to communicate.

$$\begin{aligned} P_{Alice} &= c!\langle \mathsf{enc}(v\,,\,Bob) + \mathsf{sign}(v\,,\,\mathsf{sk}(s_{Alice}))\rangle.0 \\ P_{Bob} &= c?(x).\mathsf{if}\ \mathsf{verify}(\mathsf{dec}((+_1\,x)\,,\,\mathsf{sk}(s_{Bob}))\,,\,(+_2\,x)\,,\,Alice)\ =\ \mathsf{ok}\ \mathsf{then}\ c!\langle \mathsf{ok}\rangle\ \mathsf{else}\ 0 \end{aligned}$$

Alice and Bob are identified by their public keys which are exported using active substitutions. Nonces s_{Alice} are s_{Bob} are the secrets used to generate the public/secret key pair for the participants. The roles run in the following context:

$$\nu\, s_{Alice}\,.\,\nu\, s_{Bob}\,.\,(\{\mathsf{pk}(s_{Alice})/Alice\}\mid\{\mathsf{pk}(s_{Bob})/Bob\}\mid \_)$$

Semantics The operational semantics of applied pi calculus is defined by structural equivalence, $\equiv$, and internal reduction relation, $\longrightarrow$. To reason about interactions with their environment, labelled operational semantics is most convenient, $\xrightarrow{\alpha}$ where α is an input or an output of a term that the environment observes. The definitions of these relations are given in Appendix A.1.

2.4 Refinement types for ML

In Chapters 4 to 6 we study protocols implemented in a dialect of ML, called $F\#$, where types have been refined to support logical annotations. In this section we review the syntax and semantics of the underlying core calculus, called RCF, and its implementation in the F7 typechecker. We refer to Bengtson et al. [2008b] for a detailed description of the calculus and to Bhargavan et al. [2010a] for the details on the modular verification of cryptographic protocols using F7.

2.4.1 Syntax and operational semantics of the language

RCF consists of the standard Fixpoint Calculus [Plotkin, 1985] augmented with local names, message-passing concurrency, and with refinement types. Figure 2.2 shows the full syntax of expressions we use in our formalization. Values, denoted by M, include unit, variables, pairs, constructed terms, and (possibly recursive) functions. Free variables of a term M are denoted $fv(M)$. Expressions, denoted by A, B, and e are in A-normal form; they include a standard functional core: values, function application, syntactic equality, pattern matching, **let**-bindings for sequential composition; and some concurrent constructs: name restriction, fork, and message sending and receiving on a channel. We use the directed parallel composition $A \Rsh B$; it returns the same value as B. The other concurrency and message-passing constructs are as in the pi calculus.

Our syntax slightly deviates from Bengtson et al.. The main difference is that we have recursive functions, as in F7, instead of a *fold* constructor, and we require that function values have fully-specified type annotations.

The concurrency and message passing constructs do not appear in source programs; they are used to symbolically model run-time processes (e.g. the principals running a cryptographic protocol and their adversary) and network-based communications.

Notations The source programs described in this thesis are written in a more general $F\#$ syntax that is treated as syntactic sugar for core RCF values and expressions. General expressions can be written in A-normal form by inserting **let**-bindings; for instance, $A\ B \triangleq \textbf{let}\ x = A\ \textbf{in let}\ y = B\ \textbf{in}\ x\ y$. As usual in ML, we let $\textbf{let}\ f\ x = e$ stand for $\textbf{let}\ f = \textbf{fun}\ x \rightarrow e$. We denote with $\widetilde{M}$ the tuple with n elements encoded as nested pairs $(M_1, (M_2, (\ldots, M_n)))$.

Logical annotations For specification purposes, RCF includes constructs for assuming and asserting first-order logic formulas. We let C range over formulas in a first-order logic that includes predicates over values. Formulas can be assumed (denoted **assume** C) or asserted (denoted **assert** C) by programs.

Informally, **assume**s are privileged expressions, recording for instance that a principal intends to send a message. Conversely, **assert** records that a principal believes that some logical property holds at this point. The role of **assume** and **assert** expressions is to specify, rather than to enforce, run-time properties of a program. Concretely, all formulas are erased at run-time after verification.

Operational semantics Formally an RCF expression represents a concurrent, message-passing computation, which may return a value. The state of the computation can be represented as an expression in normal form $\mathbf{S}$ (*structure*) that includes (1) a multiset of formulas that have been assumed so far; (2) a multiset of pending messages; and (3) a multiset of expressions being evaluated in parallel.

Structures and Static Safety:

$\mathcal{L} ::= \{\} \mid (\textbf{let}\ x = \mathcal{L}\ \textbf{in}\ B)$

Figure 2.2: Syntax of RCF messages, formulas and expressions

$M, N ::=$	Values
$()$	unit
x	variable
(M, N)	pair
$h\ M$	constructor
$\textbf{rec}\ f : T.\ \textbf{fun}\ x \to e$	recursive function
$e, A, B ::=$	Expressions
M	value
$M\ N$	function application
$\textbf{let}\ x = e_1\ \textbf{in}\ e_2$	sequential composition
$\textbf{let}\ (x, y) = M\ \textbf{in}\ e$	pair projection
$\textbf{match}\ M\ \textbf{with}\ h\ x\ \to e_1\ \textbf{else}\ e_2$	pattern matching (else optional)
$\textbf{assume}\ C$	assume formula
$\textbf{assert}\ C$	expect formula
$e : T$	annotated expression
$e_1 \Rsh e_2$	parallel composition
$(\nu a)e$	restriction, create new channel a
$a!M$	send M on channel a
$a?$	receive message off channel a
$C ::=$	First-order Logic Formulas
$True \mid False$	constants
$M_1 = M_2 \mid M_1 \neq M_2$	comparison
$P(M_1, \ldots, M_n)$	predicate application
$not\ C \mid C \wedge C \mid C \vee C$	boolean operators
$\forall x.C \mid \exists x.C$	first-order quantification

$$\mathbf{S} ::= (\nu a_1) \ldots (\nu a_\ell)((\prod_{i \in 1..m} \textbf{assume}\ C_i) \Rsh (\prod_{j \in 1..n} c_j!M_j) \Rsh (\prod_{k \in 1..o} \mathcal{L}_k\{e_k\}))$$

where e_k is an expression apart from a let, restriction, fork, message send, or an assumption.

Let structure $\mathbf{S}$ be *safe* if and only if, for all $k \in 1..o$ and C, if $e_k = \textbf{assert}\ C$ then the formula C is deducible from the assumed formulas, denoted $\{C_1, \ldots, C_m\} \vdash C$.

The evaluation of $\textbf{assume}\ C$ extends the current multiset of assumed formulas with C ($A \Rrightarrow A'$ denotes rewriting of A into normal form A', see Appendix A.2.1 for more details). The expression $\textbf{assert}\ C$ always reduces to unit.

$\textbf{assume}\ C \Rrightarrow \textbf{assume}\ C \Rsh ()$	(Heat Assume ())
$\textbf{assert}\ C \to ()$	(Red Assert)

We say that $\textbf{assert}\ C$ *succeeds* if, when it is evaluated, the formula C is deducible from the assumed formulas. For example, the assert in the expression $\textbf{assume}\ C; \textbf{assert}\ C$ always succeeds.

An expression is *safe* when all of its **assert**s succeed in every run.

The reduction semantics is defined in terms of a small-step relation over configurations. It contains the usual β-reduction, pattern-matching reduction and communication reduction (see

Figure 2.3: Syntax of RCF types

$P ::=$	Pretypes
$\quad unit$	$\quad$ unit type
$\quad x : T_1 \to T_2$	$\quad$ dependent function type (scope of x is T_2)
$\quad x : T_1 \times T_2$	$\quad$ dependent pair type (scope of x is T_2)
$\quad \alpha$	$\quad$ type variable
$\quad \Sigma_i(h_i : T_i \to \alpha)$	$\quad$ algebraic datatype (sum type)
$\quad \mu\alpha.P$	$\quad$ iso-recursive type (scope of α is P)
$T, U, V ::=$	Refinement Types
$\quad (x : P)\{C\}$	$\quad x$ of pretype P such that C (scope of x is C)
$E ::=$	Type Environment
$\quad \varnothing \mid E, C \mid E, x : T$	

Appendix A.2.1). Our rule for β-reduction of recursive functions is the following:

$$(\textbf{rec}\ f : T.\ \textbf{fun}\ x \to A)\ N \to A\{\textbf{rec}\ f : T.\ \textbf{fun}\ x \to A/f\}\{N/x\} \quad \text{(Red Rec Fun)}$$

Evaluation contexts E are defined as

$$E ::= [\_] \mid \textbf{let}\ x = E\ \textbf{in}\ A \mid (\nu a : T)E \mid E \Rsh B \mid B \Rsh E]$$

The expression **assert** C *succeeds* if C can be logically derived from the current multiset of assumed formulas.

2.4.2 Refinement type system

To allow static verification of the safety of RCF expressions, they are equipped with a refinement type system. The syntax of RCF types is shown in Figure 2.3.

Types are the usual ML types refined with first-order formulas. For instance, the refinement type $v : int\ \{v > 5\}$ is the type of all the expressions that, if they evaluate to a value, evaluate to an integer greater than 5. More generally, refinement types associate logical formulas with program expressions: the type of an expression A is of the form x: $P\ \{\ C\ \}$ where x binds the value of A, P is a type being refined (e.g. an ordinary ML type), and C is a formula that holds when A returns (e.g. a property of x).

Functions can also be given precise refinement types. Refinements that appear in the arguments of a function specify preconditions that must hold when the function is invoked, while the refinement of the return type specifies a postcondition that will hold when the function returns. For instance, the dependent function type $v{:}int \to w{:}int\ \{w{>}v\}$, a subtype of $int \to int$, represents functions that, when called with an integer v, may return only an integer greater than v.

Difference with the standard RCF types We use a normalized version of RCF refinement types obtained by stratifying them into pretypes (intuitively, plain ML types) and types (pretypes with a refinement). We also have recursive algebraic sum types $\Sigma_i(h_i : T_i \to \alpha)$. Datatype constructors h_i can be encoded in standard RCF using *inl*, *inr*, and *fold*.

Syntax of types Pretypes P are ML-like types extended with dependent functions, written $x : T_1 \to T_2$, and dependent pairs. To avoid an extra binder, we abbreviate $x : (x : P)\{C\} \to T_2$ to $x : P\{C\} \to T_2$. A refinement type T, of the form $x{:}P\{C\}$, is the type of expressions that return values M of pretype P such that the formula $C[M/x]$ can be derived from the log of assumed formulas. Hence, a function type can be fully written out as $x{:}(x{:}P\{C\}) \to y{:}P'\{C'\}$, where its argument has pretype P and must satisfy the precondition C, and its return value has pretype P' and is guaranteed to satisfy the postcondition C'.

A simple erasure operation converts RCF types into valid ML types (and well-typed RCF terms into well-typed ML terms).

Type environments and judgments RCF defines judgments for assigning types to expressions and for checking whether one type is a subtype of another.

Type environments E keep track of the set of assumed formulas, and typechecking ensures that every asserted formula logically holds in the current environment. (Since **assert**s may contain quantified formulas that rely on **assume**s made in concurrent expressions, it is often not possible to dynamically verify the safety of RCF expressions using run-time checks.)

The type system has the following judgments.

$E \vdash \diamond$	environment E is well-formed
$E \vdash C$	formula C holds in environment E
$E \vdash T <: T'$	T is a subtype of T' in environment E
$E \vdash e : T$	expression e has type T in environment E

An environment is well-formed if all the variables in it are well-scoped. Note, however, that the formulas in it do not have to be consistent. A formula holds in an environment if it can be deduced from the formulas in the environment. A refinement type $(x{:}P)\{C\}$ is a subtype of $(x{:}P')\{C'\}$ in an environment E, P is a subtype of P' in E and $E, x : P' \vdash C \Rightarrow C'$. For instance, $v : \mathit{int}\ \{v > 5\}$ is a subtype of int. The rest of the subtyping rules are straightforward (see Appendix A.2.2 for details).

To illustrate expression typing, we recall four typing rules, those for **assume**s and **assert**s, and those for lambda expressions and applications:

$$\text{(Typ Assume)}\quad \frac{E \vdash \diamond \quad fv(C) \subseteq dom(E)}{E \vdash \textbf{assume}\ C : (\_ : \mathit{unit})\{C\}} \qquad \text{(Typ Assert)}\quad \frac{E \vdash C}{E \vdash \textbf{assert}\ C : (\_ : \mathit{unit})\{C\}}$$

$$\text{(Typ Fun)}\quad \frac{E \vdash x : T_1 \to T_2 <: T \qquad E, f : T, x : T_1 \vdash e : T_2}{E \vdash \textbf{rec}\ f : T.(\textbf{fun}\ x \to e)\ :\ x : T_1 \to T_2} \qquad \text{(Typ App)}\quad \frac{E \vdash M : x : T_1 \to T_2 \qquad E \vdash N : T_1}{E \vdash (M\ N) : T_2\{N/x\}}$$

An **assert** C statement is well-typed in a typing environment where C logically follows from the formulas of the environment. Conversely, an **assume** C statement is always well- typed, with C as a postcondition.

A recursive function has type $x : T_1 \to T_2$ if this type is a subtype of its annotation T and its body has type T_2 in an environment extended for f and x. An application $M\ N$ has type T_2 if M has the function type type $x : T_1 \to T_2$ and N has a type which is a subtype of T_1. The rest of the typing rules are given in Appendix A.2.3.

We use the following corollary of Proposition 30 ($\to$ Preserves Types) of Bengtson et al. [2008a].

Theorem 2.1 (Subject reduction)**.** *If A is a closed expression, $E \vdash A : T$, and $A \to^* A'$, then $E \vdash A' : T$.*

Proof: By induction on the length of the derivation $A \to^* A'$. Proposition 30 of Bengtson et al. [2008a] covers the base case. □

2.4.3 Type safety

The main result of Bengtson et al. [2008b] states that if a program is well-typed, then it is safe. Moreover, if a program is well-typed in an empty environment, then it is *robustly safe*, that is, it is safe when composed with an arbitrary opponent. This theorem provides an effective method to prove protocols correct.

More in detail, we suppose that a protocol runs in presence of an active Dolev-Yao opponent. Opponents are modeled as arbitrary programs that contain no **assume**s and no **assert**s and that have access to a given public interface of the protocol. In the latest version of RCF, the code of a protocol is explicitly packaged with the imported interfaces it depends on, the interfaces it exports to the attacker, and F7 inductive definitions and theorems, to form a *refined module*.

A *module* X is a context of the form **let** $x_1 = A_1$ **in** $\ldots$ **let** $x_n = A_n$ **in** _ where $n \geq 0$ and the bound variables x_i are distinct. We let $bv(X) = \{x_1, \ldots, x_n\}$. (Following the ML syntax, we omit the keyword **in** between top-level definitions.) Modules are *independent* when the sets of predicates they define are disjoint. An *interface* I is a typing environment $\mu_1, \ldots, \mu_n$ where each μ_i is either an abstract type α_i or a variable typing $x_i : T_i$. The subtyping relation is lifted to interfaces: when $I <: I'$, each variable defined in I' must be given a supertype of the type given by I.

A module X *implements* I in E, written $E \vdash X \rightsquigarrow I$, when $E \vdash X[(x_1, \ldots, x_n)] : (x_1 : I(x_1) * \ldots * x_n : I(x_n))$.

Intuitively, a *refined module* is a triple $\mathbf{M} = (E, X, I)$ where E is the imported interface, X is the source code, and I is the exported interface of the module. More formally (after simplifying the original definition of Bhargavan et al. [2010a]), a *refined module* is a triple $\mathbf{M} = (E, X, I)$ such that there are closed formulas $\mathbf{M}^{def}$ and $\mathbf{M}^{thm}$, and a module Y where:

(1) $\mathbf{M}^{def}$ is the set of assumptions of X, in other terms $X =$ **assume** $\mathbf{M}^{def} \Rsh Y$;

(2) $\mathbf{M}^{thm}$ is a set of formulas that can be deduced from $\mathbf{M}^{def}$ (theorems);

(3) $E, \mathbf{M}^{def}, \mathbf{M}^{thm} \vdash Y \rightsquigarrow I$.

A refined module $\mathbf{M}_1 = (E_1,$ **assume** $\mathbf{M}_1^{def} \Rsh Y_1, I_1)$ *composes with* $\mathbf{M}_2 = (E_2,$ **assume** $\mathbf{M}_2^{def} \Rsh Y_2, I_2)$ iff $I_1 <: E_2$ and $\mathbf{M}_1^{def}$ and $\mathbf{M}_2^{def}$ are independent. Their *composition* $\mathbf{M}_1; \mathbf{M}_2$ is the triple $(E_1,$ **assume** $(\mathbf{M}_1^{def} \wedge \mathbf{M}_2^{def}) \Rsh Y_1[Y_2], I_2)$; it is also a refined module.

A refined module $(\varnothing, X, I)$ is *robustly safe* if and only if the expression $X[O]$ is safe for every opponent O such that $I \vdash O : unit$.

Theorem 2.2 (Robust safety by typing [Bhargavan et al., 2010a])**.** *Every refined module* $(\varnothing, X, I)$ *is robustly safe.*

Robust safety is useful for protocol security: it states that the properties of the program hold even when composed with an arbitrary active adversary that is given access to the public interface of the program.

2.4.4 Pre-defined F7 libraries

F7 comes together with a bundle of pre-defined libraries designed for implementors of cryptographic protocols. This section gives a brief overview of the libraries we rely on in our examples.

These libraries are designed as modules where inductive definitions express their logical invariants. They form composable refined modules, which allow scalable and modular code reuse and verification. Each of these modules typically has two interfaces:

– a rather abstract one, I_A, to be given to the adversary,

– and a more precise, I_M, one to be imported by other refined modules.

Given an imported interface E, one must show that both (E, M, I_A) and (E, M, I_M) are refined modules. The libraries are trusted, so that a Dolev-Yao style symbolic implementation rather than their $F\#$ implementation is typechecked against the typed interfaces.

Libraries

- *Data* defines, and provides conversion functions for, such standard datatypes as *string*, *bytes*, α *list*, α *option*. The predicate *Bytes*(x) inductively defines that x is an array of bytes that exists in a protocol run. The predicate *Pub* records if a value is known to the adversary; it is inductively defined for every data structure likely to be exchanged on a public network, and it must be extended accordingly for every defined datatype. The type of public values is **type** α *pub* = x:α {*Pub*(x)}, such abbreviations as **type** *bytespub* = *bytes pub* are defined.
- *Net* defines functions for establishing and using TCP connections;

  val *connect*: *port* → *conn*
  val *listen*: *port* → *conn*
  val *send*: *conn* → *bytespub* → *unit*
  val *recv*: *conn* → *bytespub*

- *Db* defines abstract channel-based databases;
- *Xml* defines primitives and datatypes for manipulating XML documents;
- *Crypto* defines primitives for manipulating message authentication codes (MACs), public-key signatures and encryptions (detailed below);
- *Principals* provides a convenient interface for dynamic management of principals: creation, generation, storage and retrieval of keys, as well as their leakage; a key may be leaked to model the compromise of its owner (detailed below)

The interfaces record the program invariants associated to the execution. Event predicates record progress in a protocol, for example the existence of an array of bytes b, *Bytes*(b), or that c is the result of invertible concatenation of $b1$ and $b2$, *IsConcat*(c,$b1$,$b2$). Other events must be redefined by the protocol using them (like *MACSays* explained further).

Then, functions manipulating data structures are given pre- and post-conditions to ensure that the caller code maintains those invariants. For instance, a pre-condition for sending a message x is the formula *Pub*(x). For the concatenation function for byte arrays, and its partial inverse (throwing an exception when decode fails), the programming interface for verified protocol code is

val *concat*: *b1*:*bytes* → *b2*:*bytes* → *c*:*bytes*{ *IsConcat*(*c*,*b1*,*b2*)}
val *iconcat*: *c*:*bytes* → (*b1*:*bytes* ∗ *b2*:*bytes*){ *IsConcat*(*c*,*b1*,*b2*) }

while the attacker interface is

val *concat*: *bytespub* → *bytespub* → *bytespub*
val *iconcat*: *bytespub* → *bytespub* ∗ *bytespub*

If A has to send to B some compound message $A \rightarrow B : (m1, m2)$ then A will have to show that both $m1$ and $m2$ are public, and use the following inductive rule:

$\forall b1,b2,c.\ Pub(b1) \wedge Pub(b2) \wedge Concat(c,b1,b2) \Rightarrow Pub(c)$

Crypto library

The F7 libraries interfaces can flexibly encode different cryptographic constructions. Unlike the earlier version of *RCF* using kinds and seals to model cryptography [Bengtson et al., 2008b], Bhargavan et al. [2010a] specifies and enforces invariants on the cryptographic structures via a logic model embedded in the library.

Below we outline the use of the refined interface for RSA-based public-key signatures. RSA signatures will be used as reliable evidence in Chapter 4.

The key idea is to express different cryptographic assumptions on the functions and keys (such as unforgeability of signatures, injectivity of the payload values, compromise of principals, etc.) as program invariants of the implementation.

Intuitively, the refined types for public-key signature operations let us specify the matching logical conditions between signers and verifiers. These logical conditions are used as preconditions for the sign function, and as postconditions for the signature verification function. Types are complicated by the need to take into account both compromised and uncompromised keys.

The type *key* is the type of keys defined by *Crypto*; it accounts both for keys generated by the libraries and by the adversary. Keys can be converted to and from arrays of bytes, and can be encrypted.

predicate type *pkeypreds* =
| *PrivKey* **of** *key*
| *PubKey* **of** *key*
| *PubPrivKeyPair* **of** *key* ∗ *key*
| *PrivCompKey* **of** *key*

The following three functions allow to generate a private key, derive a public key out of the private one, and leak a private key. As the **private** modifier indicates, none of them is accessible to the attacker via the library interface (but can be made so through the protocol interface). Observe that the precondition of the compromise-key function *rsa_keycomp* requires that all values encrypted with this key are public and that anything can be signed with this key.

private val *rsa_keygen*: *unit* → *sk*:*key*{ *PrivKey*(*sk*) }
private val *rsa_pub*:
sk:*key*{ *PrivKey*(*sk*)} → *k*:*key*{*PubPrivKeyPair*(*k*,*sk*) }
private val *rsa_keycomp*:
sk:*key*{ *PrivKey*(*sk*) ∧
(∀*k*,*b*. *PubPrivKeyPair*(*k*,*sk*) ∧ *CanAsymEncrypt*(*k*,*b*) ⇒ *Pub*(*b*)) ∧
(∀*b*. *SignSays*(*sk*,*b*))} →
unit { *PrivCompKey*(*sk*) }

The following two functions allow to generate signatures using a private key and verify them using the corresponding public key. In this work we slightly modify the library so that the verification function returns a Boolean (instead of raising an exception when the verification fails).

val *rsa_sign*:
sk:*key* → *b*:*bytes*
{(*PrivKey*(*sk*) ∧ *SignSays*(*sk*,*b*)) ∨ (*Pub*(*sk*) ∧ *Pub*(*b*))} →
s:*bytes*{*IsSignature*(*s*,*sk*,*b*)}
val *rsa_verify*:
vk:*key*{*PubKey*(*vk*) ∨ *Pub*(*vk*)} → *w*:*bytes* → *s*:*bytes* →
b:*bool* {*b*=**true** ⇒ ∀*sk*. *PubPrivKeyPair*(*vk*,*sk*) ⇒ *IsSignature*(*s*,*sk*,*w*)}

To resume, the predicates used by the Crypto interface are the following:

- *PrivKey*(*sk*) records that *sk* has been produced by *rsa_keygen*
- *PubKey*(*vk*) records that some key *sk* has been produced by *rsa_keygen* and that *vk* has been produced by *rsa_pub* out of *sk* (which is reflected by the following equational abbreviation).

  ∀*k*. *PubKey*(*k*) ⇔ ∃*sk*. *PubPrivKeyPair*(*k*,*sk*)

- *PubPrivKeyPair*(*vk*,*sk*) records valid pairs of verification and signing keys.
- *SignSays*(*sk*,*b*) must be defined by the protocol that relies on the key *sk* as the precondition for computing a signature and the post-condition when the verification of the signature succeeds.
- *IsSignature* records triples (sig,k,b) for which the signature function has been called or for which the verification under the public key paired with *k* succeeds; it implies either *SignSays*(*k*,*b*) or *Pub*(*k*)

$$\forall s,sk,b.\ IsSignature(s,sk,b) \wedge Bytes(s) \Rightarrow (SignSays(sk,b) \wedge PrivKey(sk)) \vee (Pub(sk) \wedge Pub(b))$$

The pre-condition of *rsa_sign* covers two cases: for a correctly generated key the plaintext to sign must be valid, and for keys which have been compromised or generated by the opponent the plaintext must be public. The pre-condition for the key of *rsa_verify* is a similar disjunction. The module also logically encodes that the produced signatures are public provided that the signed value is public.

Principals library

Module *Principals* is built on top of the module *Crypto* to manage a dynamic population of principals and their keys. It allows to generate, store, retrieve or dynamically leak keys belonging to principals. We use this library to manage keys in our multi-party game protocol. Below we briefly present the interfaces of RSA-signatures.

Type *prin* ranges over principal names and abbreviates public strings. A principal may possess several keys which can be referred to by their intended usage.

An opponent is allowed to generate a fresh pair of keys for a given principal and usage, to retrieve its public key. It can also call *leakPrivateKey* to obtain the secret key of the principal; its post-condition however tracks the compromise of the principal via predicate $Bad(p)$ which records that all keys accessible to p may have been compromised.

val *genPublicKeyPair*: *u*:*usage* → *a*:*prin* → *unit*
private val *getPublicKeyPair*: *u*:*usage* → *a*:*prin* →
  (*pk*:*key* ∗ *sk*:*key*){*PublicKeyPair*(*u*,*a*,*pk*,*sk*)}
val *getPublicKey*: *u*:*usage* → *a*:*prin* →
  pk:*key*{∃*sk*. *PublicKeyPair*(*u*,*a*,*pk*,*sk*)}
val *leakPrivateKey*: *u*:*usage* → *a*:*prin* →
  sk:*keypub*{*Bad*(*a*) ∧ (∃*pk*. *PublicKeyPair*(*u*,*a*,*pk*,*sk*))}

Predicate $SendFrom(a,u,s)$ means that the principal a intends to sign s for purpose u before sending. The module relates all principal-level predicates (in *Principals*) to the key-level predicates (in *Crypto*), for instance, *PublicKeyPair* to *PubPrivKeyPair* and *SendFrom* to *SignSays*. Key operations as well standard signing and verification operations can thus be done at the level of principals.

val *rsa_sign*: *u*:*usage* → *a*:*prin* →
  sk:*key*{*PrivateKey*(*u*,*a*,*sk*)} →
  p:*bytes*{*SendFrom*(*u*,*a*,*p*)} →
  s:*bytes*{*IsSignature*(*s*,*sk*,*p*) ∧ (*Pub*(*p*) ⇒ *Pub*(*s*))}

val *rsa_verify*: *u*:*usage* → *a*:*prin* →
  vk:*key*{*PublicKey*(*u*,*a*,*vk*)} →
  w:*bytes* → *s*:*bytes* →
  t:*bool* {*t*= **true** ⇒ (∀*sk*. *PublicKeyPair*(*u*,*a*,*vk*,*sk*) ⇒
    (*SendFrom*(*u*,*a*,*w*) ∨ *Bad*(*a*)))}

Robust safety of the trusted libraries The standard library has been proved to be robustly safe. In particular,

- the composition of *Data*, *Net* and *Crypto* is a refined module;
- the composition of *Data*, *Net*, *Crypto*, *Db*, and *Principals* is a refined module.

To show that a protocol implementation is robustly safe, we show that it is a refined module, in most cases depending on the libraries.

2.4.5 F7 implementation

The prototype typechecker, F7, is an implementation of the RCF type system that supports a significant subset of F#. In particular, it supports programs that contain type- and value-parameterized types, records, polymorphism, mutual recursion, match expressions and mutable references, but it does not, for example, support classes or objects. The typechecker takes two kinds of input files

- F# implementation files (e.g. file.fs) that mention only F# types; and
- F7 interfaces (e.g. file.fs7) with logical assumptions and RCF type annotations.

The typechecker then verifies whether an implementation is well-typed against its interface. To verify the validity of logical formulas (judgement $E \vdash C$), the typechecker can call out to any first-order logic theorem prover. It currently uses a leading SMT solver, Z3, to discharge the proof obligations. First-order logic validity is undecidable, so Z3 may fail to prove or disprove some formulas. In these cases, additional assumptions (with semi-automated proofs) are needed to verify the program.

Chapter 3

The use of logs within optimistic protocols

3.1 A cautiously optimistic approach to security

Mutual distrust in distributed computing makes enforcing system-wide security assurances particularly challenging. Common protocols perform an important number of mandatory runtime checks and allow only compliant computations to progress: in session-establishment protocols, for instance, a strong security invariant is usually enforced at every step of the run of the protocol. These runtime checks have a cost, in terms of cryptographic and networking operations; they may also conflict with other goals of the protocol, such as confidentiality.

A different approach, which we call *optimistic*, presumes instead that all involved principals are honest and well-behaved, and thus omits some runtime checks. Traces of protocol runs are stored in a secure log and can be used a posteriori to verify the compliance of each principal to its role: principals who attempt non-compliant actions will be blamed using the logged evidence. The security invariant is weaker than those achieved by more conservative protocols, but adequate for many non-critical applications.

Definition 3.1.1 (Optimistic protocol)**.** We say that a protocol is *optimistic* if its correctness and security rely on logging and auditing mechanisms as well as on the runtime checks.

An optimistic protocol run produces a log, and the only way to assess the outcome of the protocol (whether the semantics has been followed or not) is to audit (analyse) the log. To validate the design and verify the properties of such protocols, it seems crucial to answer the following questions:

- which data should be logged?
- how this data should be analysed to validate the security goals of the protocol ?

To the best of our knowledge, there is no complete and comprehensive formal study on the role of logs in optimistic protocols. In this chapter, we formally answer the questions above for two optimistic protocols:

- *value commitment* which allows to commit to a hidden value and reveal it later; our main property of interest is integrity: the committed value cannot be seamlessly changed after commitment;
- *offline e-cash* which allows to withdraw a digital coin and anonymously spend it; here again a coin is "committed" to the identity of the merchant when it is spent, and should not be spent more than once; this property must be carefully balanced to preserve the spender's anonymity.

These two schemas enjoy a similar "write-only" integrity property. The e-cash scheme can be seen as a richer extension of value commitment since it has more involved properties. To study them, we develop process algebra techniques, namely the applied pi calculus (see Section 2.3).

We first model the "ideal" semantics of these protocols, desired in a setting with no corrupted participants. Then we present a formal cryptographic implementation of this abstract semantics using audit logs, and show that this implementation is sound and safe *unless* some participant is cheating. If a participant cheats cheating, then we show that it can be discovered and blamed using the logged evidence records).

3.2 Value commitment

Protocols inherently relying on logs to establish their security properties are often based on a *commitment* scheme. A principal commits to a value kept hidden; other principals of a system cannot read this value, but have a procedure to detect any change to the value after the commitment. Distributed coin flipping is a simple protocol that illustrates commitment: suppose that A and B are not physically at the same place and want to toss a coin. Both A and B flip their own coin, exchange commitments on their results, then reveal and compare these results; A wins the toss if the two results are the same. For fairness, A's commitment should neither reveal any information to B, nor enable A to change her committed result after receiving B's.

Commitment is a building block for many protocols such as mental poker [Castellà-Roca et al., 2003], sealed bid auctions, e-voting [Chaum et al., 2004, Chaum, 2004], and online games [Jha et al., 2007]. For instance, mental poker relies on commitment to build a fair shuffling of the deck, then gradually reveal cards as the game proceeds. At the end of the game, the deck permutations used by each player can be revealed for auditing purposes.

In this section, we formally show which data should be logged for the commitment scheme. We extend the applied pi calculus with commitment datatypes and primitives, and we illustrate this extension by programming an online game. To abstract away from the possible misbehaviors of the environment, we propose a trustful and strong operational semantics for our commitment primitives. We show that our language can be compiled to the applied pi calculus, using standard cryptographic primitives, with adequate protection against an arbitrary, possibly hostile environment. We obtain an important security property stating that, for any source systems, our distributed implementation either respects the semantics of commitments or, using information stored in the logs, detects (and proves) cheating by a hostile environment.

3.2.1 A language with value commitment

To express the value commitment scheme, we extend an instance of applied pi with *committable cells*. Our extensions to the syntax of the applied pi calculus (Figure 2.1) are reported in Figure 3.2.1.

Figure 3.1: Syntax of the applied pi calculus extended with committable cells

$M,\ V$			P			$A,\ E,\ \mathcal{T}$		
	$::=$			$::=$			$::=$	
	$\mid$	$\ldots$		$\mid$	$\ldots$		$\mid$	$\ldots$
	$\mid$	$u.\mathbf{Idu}$		$\mid$	$\mathsf{newloc}\,(x, y).P$		$\mid$	$u.(p)$
	$\mid$	$u.\mathbf{Idc}(p)$		$\mid$	$\mathsf{commit}\,M\ u\,(x).P$		$\mid$	$u.(p\ M)$
	$\mid$	$u.\mathbf{Rd}(p\ M)$						

Committable cells and capabilities A cell is a memory location owned by a principal who can, *once*, commit its content to a value of its choice. In addition, the owner can pass capabilities to other principals, thereby granting these principals partial read access to the cell.

Our language features three kinds of capabilities. The *read capability* $l\,.\,\mathbf{Rd}\,(p\ M)$ is created by the owner p of the location l when it commits to a value M. Any principal can use a

read capability to read the content of the location associated to the capability. The *identity capabilities* instead partially disclose the state of a cell without actually revealing the value possibly committed. So the *committed id capability* $l\,.\,\mathbf{Idc}\,(p\,)$ proves that the location l is committed and reveals the owner p of the location. The *uncommitted id capability* $l\,.\,\mathbf{Idu}$ just asserts the identity l of the location.

The language of terms is sorted: we distinguish *marshallable values*, that include all the terms except location and channel names, and *committable values*, that include all marshallable values except those that mention committed id and read capabilities.

The state of each committable cell is represented by a process: $l.(p\,)$ denotes an uncommitted cell named l owned by p; $l.(p\,M)$ denotes the same cell once it has been committed to the committable value M. Two new kinds of processes manipulate cells. The newloc process creates a fresh, uncommitted location and binds both its unique identifier l (from $\mathcal{L}$) and its uncommitted capability in its continuation:

$$a[\mathsf{newloc}\,(x,y).P] \longrightarrow \nu\, l\,.\,(l.(a\,) \mid a[P\{^{l}/_{x}\}\{^{l\,.\,\mathbf{Idu}}/_{y}\}])$$

where l is fresh for P. The unique identifier l can then be used to commit an uncommitted cell to some committable value M:

$$l.(a\,) \mid a[\mathsf{commit}\,M\,l\,(x).P] \longrightarrow l.(a\,M) \mid a[P\{^{l\,.\,\mathbf{Rd}\,(a\;M)}/_{x}\}]$$

The commit process yields a read capability for the newly-committed cell. The sort system does not allow to communicate or store in another location the cell name l: hence, only the principal that created the cell can commit a value into it. The abbreviation newcommit creates a new committed location (where x', x'' are fresh for P):

$$p[\mathsf{newcommit}\,M\,(x).P] \stackrel{\text{def}}{=} p[\mathsf{newloc}\,(x', x'').\mathsf{commit}\,M\,x'\,(x).P]$$

Capabilities can be communicated over channels; they can also be manipulated using special functions, according to the equational theory below.

$$\mathsf{read}(x\,.\,\mathbf{Rd}\,(p\;\;v)) = v \qquad \mathsf{get\_idc}(x\,.\,\mathbf{Rd}\,(p\;\;v)) = x\,.\,\mathbf{Idc}\,(p\,)$$
$$\mathsf{get\_idu}(x\,.\,\mathbf{Idc}\,(p\,)) = x\,.\,\mathbf{Idu} \qquad \mathsf{get\_prin}(x\,.\,\mathbf{Idc}\,(p\,)) = p$$
$$\mathsf{is\_idu}(x\,.\,\mathbf{Idu}) = \mathsf{ok} \qquad \mathsf{is\_idc}(x\,.\,\mathbf{Idc}\,(p\,)) = \mathsf{ok} \qquad \mathsf{is\_rd}(x\,.\,\mathbf{Rd}\,(p\;\;v)) = \mathsf{ok}$$

The read function yields the value from read capabilities. Since the read capability is generated when committing the cell, the semantics of the source language guarantees that all reads for a given cell always return the same value. The get_prin function yields the principal that owns the cell from committed capabilities. (We could also provide get_prin from uncommitted capabilities, at some additional cost in the cryptographic implementation.) The get_idu and get_idc functions downgrade capabilities, yielding a more restrictive capability for the same cell. Hence, get_idu yields an uncommitted capability, which can be used only to identify the cell, whereas get_idc takes a read capability and hides its committed value. The language finally has functions that support dynamic typechecking of capabilities. In particular, $\mathsf{is\_idc}(x) = \mathsf{ok}$ or $\mathsf{is\_rd}(x) = \mathsf{ok}$ implies that the cell associated with x is committed.

Alternatively, commitment is modelled as standard symmetric encryption by Kremer and Ryan [2005] as part of analysis of an electronic protocol in applied pi calculus. In their work, committed values enjoy secrecy before they are opened, but they are not authenticated by their issuer.

3.2.2 Example: an online game

We can easily code the multi-party game protocol of Section 2.2.3 using committable cells. The blinded authentic messages correspond to committed id capabilities, and the non-blinded authentic messages correspond to read capabilities of the cells with the corresponding content.

The server a_0 uses channel c_i to communicate with player a_i for $i = 1..n$. We begin with the server code, given below. For simplicity, the code does not provide any error handling—execution stops when a test fails.

$$\begin{aligned} A_0 = \; & a_0[\mathsf{newloc}\,(l, result_{id}).\mathsf{newcommit}\; result_{id} + details\,(challenge). \\ & \big(c_i!\langle challenge\rangle.c_i?(promise_i).\mathsf{if}\;\mathsf{get\_prin}(promise_i) \;=\; a_i\;\mathsf{then}\,\big)_{i=1..n} \\ & \mathsf{newcommit}\; challenge + \widetilde{promise}\,(game). \\ & \big(c_i!\langle game\rangle.c_i?(move_i).\mathsf{if}\;\mathsf{get\_idc}(move_i) \;=\; promise_i\;\mathsf{then}\,\big)_{i=1..n} \\ & \mathsf{commit}\;\mathsf{winner}(\widetilde{move}, challenge)\; l\,(result).\big(c_i!\langle result\rangle.0\big)_{i=1..n}] \end{aligned}$$

In round (1), the server creates an uncommitted cell l for storing the outcome of the game, and a readable cell *challenge* that provides the identifier for l and the (unspecified) details of the game. Upon receiving each player's response, the server authenticates it as a committed capability from that player. In round (2), the server creates a second committed cell that binds the challenge to the received commitments from all players. Upon receiving each player's second response, the server correlates it as the read capability associated with their first response. In round (3), the server has all the players' information: it resolves the game and finally commits the cell l to the published result of the game (which may include, for instance, selected information from the players' moves). We omit the code for the function winner that computes this result.

The code for the players performs symmetric operations:

$$\begin{aligned} A_i = a_i[\, & c_i?(challenge).\mathsf{if}\;\mathsf{get\_prin}(\mathsf{get\_idc}(challenge)) \;=\; a_0\;\mathsf{then} \\ & \mathsf{newcommit}\; z_i\,(move_i).c_i!\langle\mathsf{get\_idc}(move_i)\rangle. \\ & c_i?(game).\mathsf{if}\;\mathsf{valid\_game}\,(\,game\,,\,challenge\,,\,move_i\,)\;\mathsf{then} \\ & \quad c_i!\langle move_i\rangle.c_i?(result_i).\mathsf{if}\;\mathsf{no\_cheat}\,(\,result_i\,,\,\mathsf{read}(game)\,)\;\mathsf{then}\;P_i] \end{aligned}$$

In round (1), after receiving the challenge, each player confirms its validity, for instance by checking that it is a genuine readable capability from a_0, then it selects a move and sends back its commitment. In round (2), after receiving all commitments, the player correlates them to the challenge and verifies that its own commitment is recorded (using for instance valid_game) then it releases its move in clear. In round (3), the player checks the outcome of the game and verifies a posteriori that the server followed the rules (using for instance no_cheat). The tests are defined as follows:

$$\begin{aligned} \mathsf{valid\_game}\,(\,x_1\,,\,x_2\,,\,x_3\,) & \stackrel{\text{def}}{=} \; +_1(\mathsf{read}(x_1)) \;=\; x_2\;\mathsf{and}\;\mathsf{get\_idc}(x_3) \in +_2(\mathsf{read}(x_1)) \\ \mathsf{no\_cheat}\,(\,x\,,\,y\,) & \stackrel{\text{def}}{=} \; \mathsf{get\_idu}(\mathsf{get\_idc}(x)) \;=\; +_1(y)\;\mathsf{and}\;\mathsf{get\_idc}(x) \in +_2(y) \end{aligned}$$

Guarantees offered to the players We distinguish *language level* guarantees, enforced by the abstract semantics of locations, and *application level* guarantees, relying on high-level, application-specific checks on top of the language semantics. For each kind of guarantees, we also distinguish between immediate (conservative) and deferred (optimistic) enforcement. For instance, enforcement may be deferred until the content of a cell becomes readable.

As an illustration of immediate language-level checks, committed values offer basic authentication guarantees to the participants. For instance, each player has the privilege to choose its moves, and the move is securely attributed to the player even if the communication channels c_i are unprotected; participants can also check this attribution later.

To protect application integrity, the code must perform sufficient checks before proceeding with the game. Systematic testing of the owner identities for the received capabilities avoids unauthorized, possibly non-accountable, participants. Some checks are immediate, e.g. testing if two capabilities are associated to the same location; other checks that depend on the commitment semantics are delayed. In the example, players are guaranteed that they all get the same result (if any) for any given game, since they must get the same location read capability, but it

is up to the application code to correlate the received read capability to the initial uncommitted capability.

At the same time, the applicative logic of our protocol guarantees that, even if the server is willing to leak information to the other players, those players cannot get that information before committing to their own moves.

3.2.3 Distributed cryptography implementation

The target language is an instance of applied pi, with standard (symbolic) cryptographic primitives and data structures but without ad-hoc rules or constructs for locations.

We rely on a cryptographic hash function, denoted h, and a public-key signature mechanism satisfying the equation $\mathsf{verify}(v\,,\,\mathsf{sign}(v\,,\,\mathsf{sk}(m))\,,\,\mathsf{pk}(m)) = \mathsf{ok}$. The functions $\mathsf{sk}(m)$ and $\mathsf{pk}(m)$ generate a pair of secret/public keys from a nonce m. All other data constructors (including rd, idc, idu, prin introduced below) admit projection functions $func_i(func(x_1\,,\,\ldots\,,\,x_n)) = x_i$.

To every principal p, we associate a key pair and export its public key tagged with constructor prin using an active substitution of the form $\{\mathsf{prin}(\mathsf{pk}(m_p))/p\}$.

Cryptographic implementation of capabilities We compile the capabilities associated to a location $l.(p\ V)$ as follows:

$l\,.\,\mathbf{Rd}\,(p\ \ V)$	$\mathsf{rd}(p\,,\,s\,,\,[\![V]\!]\,,\,w)$
$l\,.\,\mathbf{Idc}\,(p\,)$	$\mathsf{idc}(p\,,\,\mathsf{h}(s) + \mathsf{h}(s + [\![V]\!])\,,\,w)$
$l\,.\,\mathbf{Idu}$	$\mathsf{idu}(\mathsf{h}(p + \mathsf{h}(s)))$

where $p = \mathsf{prin}(\mathsf{pk}(m_p))$ is the owner's public key, s is a fresh value used as a seed, and $w = \mathsf{sign}(\mathsf{h}(s) + \mathsf{h}(s + [\![V]\!])\,,\,\mathsf{sk}(m_p))$ signs the committed value $[\![V]\!]$.

A read capability is a tagged tuple that includes these elements. A committed id capability is a tagged tuple that provides p and verifiable evidence of the commitment without actually revealing $[\![V]\!]$. To this end, it includes both a hash of the committed value, first concatenated with the seed s, to protect against brute force attacks, yielding $\mathsf{h}(s + [\![V]\!])$, and the hash $\mathsf{h}(s)$, to enable the receiver to correlate the owner and signature with a previously-received uncommitted id capability by recomputing the identifier $\mathsf{h}(p + \mathsf{h}(s))$. An uncommitted id capability just includes this unique location identifier, which may be compared to other capabilities and, later, associated with p and s. The receiver can compute committed capabilities from read capabilities, and uncommitted capabilities from committed capabilities, but not the converse.

The signature w authenticates read and committed id capabilities, binding their content to the owner's key $\mathsf{sk}(m_p)$. Their receiver can extract p and $\mathsf{h}(s) + \mathsf{h}(s + [\![V]\!])$ from these tagged tuples and use them to verify w. When the signature is valid, the public key identifies the owner of the location associated to the capability.

Detection of multiple commitments In a typical run, an honest principal receives a commitment to some value from the principal p, say $\mathsf{idc}(p\,,\,v_1 + v_2\,,\,w)$, and later the value itself, say $\mathsf{rd}(p\,,\,s\,,\,z\,,\,w')$. The receiver can easily check that the two capabilities refer to the same location, by testing $\mathsf{h}(s) = v_1$, and verify the two signatures $w = \mathsf{sign}(v_1 + v_2\,,\,\mathsf{sk}(m_p))$ and $w' = \mathsf{sign}(\mathsf{h}(s) + \mathsf{h}(s + z)\,,\,\mathsf{sk}(m_p))$. If these tests succeed, then the receiver can check whether $v_2 = \mathsf{h}(s + M)$: if the test fails, the principal p can be convicted of multiply committing the location identified by $\mathsf{h}(p + \mathsf{h}(s))$.

In preparation for the translation, we introduce functions that operate on tuples representing capabilities in the target language. For instance, the function read implements source-language

reads as a projection, and check_idc verifies the seal of committed ids.

$$\begin{aligned}
\mathsf{read}(x) &\stackrel{\text{def}}{=} \mathsf{rd}_3(x)\\
\mathsf{get_idc}(x) &\stackrel{\text{def}}{=} \mathsf{idc}(\mathsf{rd}_1(x)\,,\,\mathsf{h}(\mathsf{rd}_2(x)) + \mathsf{h}(\mathsf{rd}_2(x) + \mathsf{rd}_3(x))\,,\,\mathsf{rd}_4(x))\\
\mathsf{check_idc}(x) &\stackrel{\text{def}}{=} \mathsf{verify}(\mathsf{idc}_2(x)\,,\,\mathsf{idc}_3(x)\,,\,\mathsf{prin}_1(\mathsf{idc}_1(x))) = \mathsf{ok}\\
\mathsf{get_idu}(x) &\stackrel{\text{def}}{=} \mathsf{idu}(\mathsf{h}(\mathsf{idc}_1(x) + (+_1\,\mathsf{idc}_2(x))))
\end{aligned}$$

In general, inconsistent capabilities may be scattered in the whole system. To detect such inconsistencies and reliably blame cheating principals, a compiled system logs all the committed capabilities generated or received by honest principals by sending them over the channel *log* to the following resolution process R. For technical convenience, we introduce the notation $Q\,(\,y_1\,)$ for the verification process, parameterized by a first capability y_1 and that becomes deterministic after inputting its a second capability y_2.

$$\begin{aligned}
Q\,(\,y_1\,) \stackrel{\text{def}}{=}\ & \mathit{log}?(y_2).\mathsf{if}\ \mathsf{check_idc}(y_1)\ \mathsf{and}\ \mathsf{check_idc}(y_2)\ \mathsf{then}\\
& \quad \mathsf{if}\ \mathsf{get_idu}(y_1) = \mathsf{get_idu}(y_2)\ \mathsf{and}\ \mathsf{idc}_2(y_1) \neq \mathsf{idc}_2(y_2)\ \mathsf{then}\ \mathit{bad}!\langle\mathsf{get_prin}(y_1)\rangle\\
R \stackrel{\text{def}}{=}\ & (\mathsf{repl}\ \mathit{log}?(y_1).Q\,(\,y_1\,))\mid(\mathsf{repl}\ \mathit{log}!\langle\mathsf{None}\rangle)
\end{aligned}$$

This resolution process repeatedly reads pairs of Idc capabilities over the *log* channel and tests them for inconsistencies, as described above. If cheating is detected, the principal is blamed on channel *bad*. To model the fact that any resolution continuation can be discarded, we add a replicated output of the harmless message None on *log* within resolution process.

The resolution process acts as an external judge auditing the compiled system, and the data sent over the channel *log* as a secure audit trail. Since all messages on *log* are replicated, log entries cannot be erased or modified by a malicious principal, and every principal may run its own copy of process R. At the same time, a malicious principal cannot forge capabilities that would accuse an honest principal, as it cannot produce a valid seal associated with the honest principal.

In Section 3.2.5 we establish a correspondence between source systems and their cryptographic implementation. In the implementation, when a resolution process receives a first capability, its further behaviour still depends on the second capability to be received; this intermediate "waiting" state is not represented in the source level. To leverage this we extend source systems to include a *resolution store* $\mathcal{T} = \prod_{0<i<n} \mathsf{resolving}(H_i)$ where each of H_i is either a (possibly fresh) low-level term sent by the adversary, or an exported local term containing names restricted in $\mathcal{N}$, and where each of $\mathsf{resolving}(H_i)$ denotes a resolution process that has already input H_i and is waiting for a second input. The store $\mathcal{T}$ is passive; in particular it can be discarded at any time, as defined by the reduction rules below:

$$\frac{\mathsf{adversary\ knows}\ H}{0 \longrightarrow \mathsf{resolving}(H)}\ \textsc{AddRes} \qquad \frac{}{\mathsf{resolving}(H) \longrightarrow 0}\ \textsc{DelRes}$$

Translation of initial configurations The cryptographic implementation is obtained by translating the source configurations. Protocol descriptions can be expressed as initial configurations of a source system that do not contain, or refer to, locations and capabilities; these are created later, during the run of the protocol. We describe the translation of such configurations; a full treatment of capabilities and locations is deferred to Section 3.2.4.

Our main translation function for configurations is denoted as $[\![\![\,]\!]\!]$ and uses auxiliary recursive translations for configurations, terms, and processes, all denoted as $[\![\,]\!]$.

The translation is a homomorphism over terms and over most systems.

$$[\![x]\!] = x \qquad [\![c]\!] = c \qquad [\![\mathit{func}(M_1\,,\,\ldots\,,\,M_n)]\!] = \mathit{func}([\![M_1]\!]\,,\,\ldots\,,\,[\![M_n]\!])$$
$$[\![A_1 \mid A_2]\!] = [\![A_1]\!] \mid [\![A_2]\!] \qquad [\![\nu\, u\,.\,A]\!] = \nu\, u\,.\,[\![A]\!] \qquad [\![\{M/x\}]\!] = \{[\![M]\!]/x\}$$

Figure 3.2: Translation of processes

$$[\![\mathsf{newloc}\,(x,y).P]\!]_a = \nu\, s_{l'}\,.\,\nu\, c_l\,.\,\tau.(c_l!\langle\mathsf{None}\rangle \mid [\![P]\!]_a\,\{^{c_l}/_{c_x}\}\,\{^{s_{l'}}/_{s_x}\}\,\{^{\mathsf{h}(a+\mathsf{h}(s_{l'}))}/_{l}\}\,\{^{\mathsf{idu}(l)}/_{y}\})$$

$$[\![\mathsf{commit}\ V\ x\ (x').P]\!]_a = c_x?(y).([\![P]\!]_a \mid \mathsf{repl}\ log!\langle\mathsf{idc}(a\ ,\ v_x\ ,\ w_x)\rangle)$$
$$\{^{\mathsf{h}(s_x)+\mathsf{h}(s_x+[\![V]\!])}/_{v_x}\}\,\{^{\mathsf{sign}(v_x\,,\,\mathsf{sk}(m_a))}/_{w_x}\}\,\{^{\mathsf{rd}(a\,,\,s_x\,,\,[\![V]\!]\,,\,w_x)}/_{x'}\}$$

$$\begin{array}{l}
\mathsf{parse_c}\ x\ P = \\
\quad \mathsf{if\ is_idu}(x) = \mathsf{ok\ then}\ P \\
\quad\quad \mathsf{else\ if\ is_prin}(x) = \mathsf{ok\ and\ is_pk}(x) = \mathsf{ok\ then}\ P \\
\quad\quad\quad \mathsf{else\ if\ is_pair}(x) = \mathsf{ok\ then\ parse_c}\ (+_1\ x)\ (\mathsf{parse_c}\ (+_2\ x)\ P) \\
\quad\quad\quad\quad \mathsf{else}\ r!\langle\mathsf{None}\rangle \\
\mathsf{parse}_1\ x\ P = \\
\quad \mathsf{if\ is_rd}(x) = \mathsf{ok\ then} \\
\quad\quad \mathsf{if\ check_idc(get_idc}(x)) \mathsf{\ then\ parse_c\ read}(x)\ P \mathsf{\ else}\ r!\langle\mathsf{None}\rangle \\
\quad\quad \mathsf{else\ if\ is_idc}(x) = \mathsf{ok\ then\ if\ check_idc}(x) \mathsf{\ then}\ P \mathsf{\ else}\ r!\langle\mathsf{None}\rangle \\
\quad\quad\quad \mathsf{else\ if\ is_pair}(x) = \mathsf{ok\ then\ parse}_1\ (+_1\ x)\ (\mathsf{parse}_1\ (+_2\ x)\ P) \\
\quad\quad\quad\quad \mathsf{else\ parse_c}\ x\ P \\
\mathsf{parse}_2\ x\ = \\
\quad \mathsf{if\ is_rd}(x) = \mathsf{ok\ then\ repl}\ log!\langle\mathsf{get_idc}(x)\rangle \\
\quad\quad \mathsf{else\ if\ is_idc}(x) = \mathsf{ok\ then\ repl}\ log!\langle x\rangle \\
\quad\quad\quad \mathsf{else\ if\ is_pair}(x) = \mathsf{ok\ then}\ (\mathsf{parse}_2\ (+_1\ x)\ \mid \mathsf{parse}_2\ (+_2\ x)\ ) \\
\mathsf{parse}\ x\ P = \mathsf{parse}_1\ x\ (P \mid \mathsf{parse}_2\ x\ )
\end{array}$$

$$[\![c!\langle M\rangle.P]\!]_a = c!\langle[\![M]\!]\rangle.[\![P]\!]_a$$

$$[\![c?(x).P]\!]_a = \nu\, r\,.\,(c?(x).\mathsf{parse}\ x\ [\![P]\!]_a \mid \mathsf{repl}\ (r?(\_).c?(x).\mathsf{parse}\ x\ [\![P]\!]_a))$$

$$[\![\mathsf{if}\ M\ =\ M'\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2]\!]_a = \mathsf{if}\ [\![M]\!]\ =\ [\![M']\!]\ \mathsf{then}\ [\![P_1]\!]_a\ \mathsf{else}\ [\![P_2]\!]_a$$

Let $\mathcal{A}$ the set of principals running a process in the system and $\mathcal{E}$ the set of other (possibly dishonest) principals whose names occur in the system ($\mathcal{E} = \mathcal{P} \cap \mathrm{fn}(A) \setminus \mathcal{A}$).

For each principal $a \in \mathcal{A}$, the translation creates a secret seed m_a used to generate the pair of secret/public keys of the principal. The public key is published using an active substitution, while the process run by the principal is compiled within the scope of the private seed m_a used for signing. Similarly, the translation includes active substitutions E that records, for each principal $e \in \mathcal{E}$, a public key $\mathsf{pk}(m_e)$ and an associated secret H_e. Resolution store translates into partial resolution processes Q.

$$[\![a[P]]\!] = \nu\, m_a\,.\,([\![P]\!]_a \mid \{\mathsf{prin}(\mathsf{pk}(m_a))/a\})$$
$$E = \textstyle\prod_{e\in\mathcal{E}}(\{\mathsf{prin}(\mathsf{pk}(m_e))/e\} \mid \{H_e/m_e\})$$
$$[\![\mathsf{resolving}(H)]\!] = Q\ (\ H\ )$$

The main translation applies the auxiliary translation to the configuration, publishes the public keys of possibly dishonest principals E, and also spawns a replicated resolution server R:

$$[\![A]\!] = [\![A]\!] \mid R \mid E$$

The translation of processes is given in Figure 3.2. (We omit the homomorphic clauses for 0, $P_1 \mid P_2$, $\mathsf{repl}\ P$, and $\nu\, c\,.\,P$).

The translation of newloc creates a fresh location seed s'_l and a local channel c_l (with a message None, recording that the location is uncommitted), and substitutes c_l for c_x, s'_l for s_x

and the idu capability for y in the continuation. We let $\tau.A$ abbreviate if $n = n$ then A, a process that reduces to A in one silent step. To match our main theorem's statement, every translation process should make at least one labelled transition, so we force the translation to make a silent transition first, using τ construct.

The translation of commit can proceed only if the location has not been previously committed (the message on c_x provides mutual exclusion); it then substitutes the rd capability for x' in the continuation code. It also generates the corresponding idc capability for the location and logs it by sending it to the resolution protocol.

The parse function filters any received value received over channels. Terms in our source systems are well-sorted. In the translation of the input, we encode dynamic sorting. If the value is tagged with rd or idc, then it might (or not) be a valid capability, depending on the validity of its embedded signature: valid capabilities are passed to the continuation, while the associated idc should be sent to the resolution protocol. To make this atomic, we separate verifications from logging, so that logging is only done if the entire message is well-formed. Filter parse_1 checks if its message argument is marshallable. We also check that a read capability only contains committable terms: filter parse_c checks if its argument is committable. Filter parse_2 logs the committed parts of the message. If the value is tagged with idu, then it is always passed to the continuation. If the value is tagged with prin, then since the set of known principals $\mathcal{P}$ is static and parse must check whether the received value is a public key known by the system S. We use predicate $\mathsf{is\_pk}(x)$ which succeeds if x is tagged with pk and if the context contains an active substitution $\{x/p\}$ for some principal $p \in \mathcal{A} \cup \mathcal{E}$. For compound data, here pairs, each element is separately handled using parse_1 then parse_2. Other values, as well as non-valid committed capabilities, are silently discarded. In the translation of an input, we assume that the channel r is fresh for $[\![P]\!]_a$, and use this channel to loop after discarding such values.

3.2.4 Model and translation of environment interactions

Our main results relate the behaviour of source systems to their cryptographic implementations (see Section 3.2.5). We describe the behaviour of systems using labelled semantics that explicitly capture all possible interactions between a system composed of honest principals and an abstract environment composed of potentially hostile principals. We define a labelled semantics for our source language, and we use the labelled semantics of the applied pi calculus for the implementation. To maintain the committable-cell invariants, our source semantics keeps track of the capabilities exported to the environment and of the partial knowledge acquired when receiving capabilities from the environment. We then extend our translation from initial configurations to any such reachable configuration.

Extended location states and capabilities We extend the committable location states in the source language. We use overlapping syntaxes for capabilities appearing in values, in transition labels, and in the processes representing the state of the cells. Their general form is $l\,.\,Cap\,(\,[\,p\,]\,[\,H\,]\,[\,V\,]\,)$, where l is the location identifier; $Cap \in \{0, \mathbf{Idu}, \mathbf{Idc}, \mathbf{Rd}\}$ is a capability tag; p is a principal name; H ranges over terms of the target language; and V is a value of the source language. (This syntaxes extend those given in Section 3.2.1 for capabilities and location states, with $l.(a\,M) = l\,.\,0\,(a\;M)$). The fields p, H, and V are optional. The presence of a value V indicates that the location is committed to this value. The term H plays no role in the source language, but is technically convenient in its translation: it enables us to represent any reachable state of our implementation as the translation of a source system.

The interpretation of Cap depends on the principal p that owns the location. If a location is owned by $a \in \mathcal{A}$, then Cap represents the most permissive capability *sent to* the environment (and H is omitted), with $Cap = 0$ when no capabilities have been exported so far. If a location is owned by $e \notin \mathcal{A}$, then Cap represents the most permissive capability *received from* the environment (and H records some opaque cryptographic value in its received representation).

Ordering capabilities We formalize the notion of "more permissive capability" by defining a preorder $\preceq$ on capabilities. Intuitively, $C \preceq C'$ holds if C and C' have compatible contents and C can be derived from C' using the equational theory. We also introduce a special capability $\bot$ that represents the absence of knowledge on a location. The order is defined by the axioms below:

$$\bot \preceq 0\ ct \qquad 0\ ct \preceq \mathbf{Idu}\ ct \qquad \mathbf{Idu}\,\mathsf{f}_u\,(ct) \preceq \mathbf{Idc}\ ct \qquad \mathbf{Idc}\,\mathsf{f}_c\,(ct) \preceq \mathbf{Rd}\ ct$$
$$Cap\ (p\ H) \preceq Cap\ (p\ H\ V)$$

where ct is any fixed contents and f_u and f_c are fixed functions that rewrite H in ct. These functions are defined as:

$$\mathsf{f}_u(a\ \ V) = (a\,) \qquad \mathsf{f}_u(e\ H\ V) = (e\ \mathbf{h}(+_1(H)+e))$$
$$\mathsf{f}_c(a\ \ V) = (a\ \ V) \qquad \mathsf{f}_c(e\ H\ V) = (e\ \mathbf{h}(H)+\mathbf{h}(H+[\![\,V\,]\!])\ V)$$

We write $C \curlyvee C'$ for the sup of C and C' with respect to $\preceq$, when it exists.

Normal form We say that a source system is in *normal form* when it is of the form

$$S \ = \ \nu\mathcal{N}\,(\textstyle\prod_{l\in\mathcal{L}} l\,.\,C_l \mid \prod_{a\in\mathcal{A}} a[P_a] \mid \phi)$$

for some finite sets of names $\mathcal{N}$, $\mathcal{L}$, and $\mathcal{A}$ and active substitutions ϕ. Every initial configuration can be written in normal form (with $\mathcal{L} = \varnothing$) using structural equivalence.

Definition 3.2.1. A system S is *well-formed* when it is structurally equivalent to a normal form such that if l is a location name within S then $l \in \mathcal{L}$ and l occurs only

(1) in terms $l.C$ such that: (a) if $\mathsf{get\_prin}(l\,.\,C_l) \in \mathcal{A}$, then C and C_l are owned by the same principal and if C has a value, then C_l has the same value; and
(b) if $\mathsf{get\_prin}(l\,.\,C_l) \notin \mathcal{A}$, then $C \preceq C_l$ (informally, for a cell owned by the environment, the system cannot have capabilities more permissive than those received);

(2) in subprocesses $\mathsf{commit}\ M\ l\,(x).P$ of P_a when $a = \mathsf{get\_prin}(l\,.\,C_l)$;

(3) in $\mathcal{N}$ when $\mathsf{get\_prin}(l\,.\,C_l) \in \mathcal{A}$ and $C_l = 0\ ct$.

In the labelled semantics below, we require that the initial and final systems and the label be well-formed. We define labelled transitions $A \xrightarrow{\alpha} A'$ between source systems on top of an auxiliary relation $C \xrightarrow{\gamma} C'$ between capabilities.

Labelled transitions on capabilities Input/output actions with the environment can affect the state of memory cells. To model these updates compositionally we define a labelled transition semantics between capabilities.

$$\frac{}{C \xrightarrow{!\,C'} C \curlyvee C'} \qquad \frac{C' \preceq C \wedge \mathsf{prin\_of}(C') \in \mathcal{A}}{C \xrightarrow{?\,C'} C} \qquad \frac{\mathsf{prin\_of}(C') \notin \mathcal{A}}{C \xrightarrow{?\,C'} C \curlyvee C'}$$

The label $!\,C'$ records that the capability C' is exported to the environment: the outcome of the transition $C \curlyvee C'$ is an updated record of the most permissive exported capability. The label $?\,C'$ records that the capability C' is imported from the environment. There are two import rules, depending on the owner of C'. If the owner is in $\mathcal{A}$, then the capability refers to a location which is part of the system, so the environment can send back at most capabilities that can be derived from those exported by the system, hence the $C' \preceq C$ condition. On the contrary, if the owner is not in $\mathcal{A}$, the environment can send any capability, provided that the capability is compatible with the partial knowledge that the system already has, i.e. that $C \curlyvee C'$ exists.

Labelled transitions on systems The labelled semantics for systems is adapted from the one for the applied pi calculus. We point out the novelties, and refer to Appendix B.1.6 for the full semantics.

The labelled semantics has silent steps for all system reductions, including the location-specific reductions described in Section 3.2.1. The axioms for input and output are recalled below.

$$a[c!\langle M\rangle.P] \xrightarrow{c\,!\,M} a[P] \qquad a[c?(x).P] \xrightarrow{c?M} a[P\{^{M^\sharp}/_x\}]$$

When a capability is received, the rule substitutes in a capability value $M^\sharp$ obtained from the capability label M by erasing information used only to update the cell state. Erasing is defined as follows:

$$l\,.\,\mathbf{Idu}\,(p\ H)^\sharp = l\,.\,\mathbf{Idu} \qquad l\,.\,\mathbf{Idc}\,(p\ H\ V)^\sharp = l\,.\,\mathbf{Idc}\,(p\ ) \qquad l\,.\,\mathbf{Rd}\,(p\ H\ V)^\sharp = l\,.\,\mathbf{Rd}\,(p\ \ V)$$

Capability labels and derived capabilities have the same translation $[\![M^\sharp]\!] = [\![M]\!]$.

The context rules below ensure that the communication of capabilities is reflected in the state of the cells of the system; the function $\mathsf{locs}(l, M)$ computes the most permissive among the capabilities for cell l that occur in the transmitted capability (possibly within another capability). If the simultaneous commitments are incompatible, their sup does not exist and the update of memory cells is impossible.

$$\frac{A \xrightarrow{c\,!\,M} A' \wedge C_0 \xrightarrow{!\,\mathsf{locs}(l,M)} C_1}{l\,.\,C_0 \mid A \xrightarrow{c\,!\,M} l\,.\,C_1 \mid A'} \qquad \frac{A \xrightarrow{c?M} A' \wedge C_0 \xrightarrow{?\mathsf{locs}(l,M)} C_1}{l\,.\,C_0 \mid A \xrightarrow{c?M} l\,.\,C_1 \mid A'}$$

$$\begin{aligned}
\mathsf{locs}(l, u) &= \bot \\
\mathsf{locs}(l, l\,.\,Cap\,(p\ H\ V)) &= Cap\,(p\ H\ V) \text{ when } Cap \in \{\mathbf{Idu}, \mathbf{Idc}\} \\
\mathsf{locs}(l, l\,.\,\mathbf{Rd}\,(p\ H\ V)) &= \mathbf{Rd}\,(p\ H\ V) \curlyvee \mathsf{locs}(l, V) \\
\mathsf{locs}(l, M_1 + M_2) &= \mathsf{locs}(l, M_1) \curlyvee \mathsf{locs}(l, M_2)
\end{aligned}$$

We equate $l\,.\,\bot \mid A$ to A, so that the input rule covers the case of an input carrying fresh, unknown locations from the environment. (The resulting configuration must be well-formed, which excludes the introduction of a fresh location state for l if one already exists in the system.) We impose the following well-formedness conditions on labels: the target term H, the principal in uncommitted capabilities, and the value in committed capabilities, appear iff the transition is an input and the capability is owned by $e \notin \mathcal{A}$.

Example of transitions in the source language Consider the third round of the game of Section 3.2.2, with two honest players a_1 and a_2 and an external, untrusted principal $e_0 \notin \mathcal{A}$ running the server. A simplified configuration of this system can be written

$$A' = l\,.\,\mathbf{Idu}\,(e_0\ H) \mid a_1[c_1?(x_1).P_1] \mid a_2[c_2?(x_2).P_2]$$

where l is the uncommitted cell pre-allocated by e_0 to store the winning move. (Here $H = \mathsf{h}(e_0 + \mathsf{h}(s))$ for some secret s created by e_0.) We have possible input transitions on channels c_1 and c_2, to notify the winning move to each of the players. The first transition may be:

$$A' \xrightarrow{c_1?l\,.\,\mathbf{Rd}\,(e_0\ s\ \mathbf{11})} l\,.\,\mathbf{Rd}\,(e_0\ s\ \mathbf{11}) \mid a_1[P_1\{^{l\,.\,\mathbf{Rd}\,(e_0\ \mathbf{11})}/_{x_1}\}] \mid a_2[c_2?(x_2).P_2]$$

which triggers the final process P_1 with a read capability for l substituted for x_1, carrying the game result (here 11). At the same time, the state for l is updated by the third capability-transition rule, since $\mathbf{Idu}\,(e_0\ H) \curlyvee \mathbf{Rd}\,(e_0\ s\ \mathbf{11}) = \mathbf{Rd}\,(e_0\ s\ \mathbf{11})$. Conversely, for instance, transitions with a label that attributes l to a_1 instead of e_0 are disabled. At this stage, the configuration

records the commitment on l, so the only subsequent input transition $A'' \xrightarrow{c_2?l\,.\,C'} A'''$ carrying a read capability C' for l must be such that $\mathbf{Rd}\,(e_0\; s\; \mathbf{11}) \preceq C'$ (by the third capability-transition rule), that is, $C' = \mathbf{Rd}\,(e_0\; s\; \mathbf{11})$. This guarantees that the second player gets exactly the same result as the first one.

Relating the reduction-based and labelled semantics for the source language The labelled semantics precisely characterizes the interactions between a system and an arbitrary environment. Given two systems A and E consisting of principals in $\mathcal{A}$ and $\mathcal{E}$, respectively, if $E \mid A \longrightarrow^* S$ then there exist two such systems A' and E' and transitions $A \xrightarrow{\phi} A'$ such that $S \equiv \nu\mathcal{N}.(E' \mid A')$, where $\mathcal{N}$ is the set of names exported in the labels of ϕ. Conversely, for all systems A and transitions $A \xrightarrow{\phi} A'$, there exists a system E' and reductions $E \mid A \longrightarrow^* \nu\mathcal{N}.(E' \mid A')$.

Translation of extended location states and capabilities We extend the translation of Section 3.2.3 to cover all configurations reachable by transitions from initial configurations. This extended translation is inductively defined for all well-formed configurations in normal form, using the clauses of Section 3.2.3 plus the rules below for location states and capabilities.

We extensively rely on active substitutions [Abadi and Fournet, 2001] with the following naming conventions: for a location l, c_l denotes the local channel that contains the state of the location, s_l the secret seed, v_l the hidden value, and w_l the seal. We define two extended processes that compute and log identifiers, commitment values, and seals for a location owned by a given principal p using active substitutions.

$$\varphi(M_1, M_2)_p = \{\mathsf{h}(p + M_1)/l\} \mid \varsigma(M_1, M_2)_p$$
$$\varsigma(M_1, M_2)_p = \{M_1 + M_2/v_l\} \mid \{\mathsf{sign}(v_l\,,\, \mathsf{sk}(m_p))/w_l\} \mid \mathsf{repl}\; log!\langle \mathsf{idc}(p\,,\, v_l\,,\, w_l)\rangle$$

We first translate locations owned by honest principals $a \in \mathcal{A}$. The translation implements these locations by sending the location state on the local channel c_l, activating the relevant substitutions, creating a fresh secret and, for committed locations only, running a replicated log entry:

$$[\![l\,.\,0\,(a\,)]\!] = [\![l\,.\,\mathbf{Idu}\,(a\,)]\!] = c_l!\langle \mathsf{None}\rangle \mid \{\mathsf{h}(a + \mathsf{h}(s_l))/l\} \mid \nu\, s\,.\,\{s/s_l\}$$
$$[\![l\,.\,0\,(a\;\; V)]\!] = [\![l\,.\,\mathbf{Idc}\,(a\;\; V)]\!] = [\![l\,.\,\mathbf{Rd}\,(a\;\; V)]\!] = \varphi(\mathsf{h}(s_l), \mathsf{h}(s_l + [\![V]\!])_a \mid \nu\, s\,.\,\{s/s_l\}$$

We also translate locations owned by principals $e \notin \mathcal{A}$ whose capabilities have been previously received by some principals in $\mathcal{A}$. The translation records partial knowledge of these locations, in the form of active substitutions plus, for committed locations only, a replicated log entry. The form of the terms in these substitutions reflect the test that processes in $\mathcal{A}$ have successfully performed before accepting these values, e.g. that the seal is a well-formed signature from e.

$$[\![l\,.\,\mathbf{Idu}\,(e\; H)]\!] = \{H/l\}$$
$$[\![l\,.\,\mathbf{Idc}\,(e\;(\;M' \;+\; M''\;)\; V)]\!] = \varphi(M', M'')_e$$
$$[\![l\,.\,\mathbf{Rd}\,(e\; M\;\; V)]\!] = \{M/s_l\} \mid \varphi(\mathsf{h}(M), \mathsf{h}(M + [\![V]\!])_e$$

In a well-formed system, there is a location state for every capability that occurs in the system. Accordingly, the translation of capabilities relies on the active substitutions introduced by the translation of location states, as follows:

$$[\![l\,.\,\mathbf{Idu}]\!] = \mathsf{idu}(l) \quad [\![l\,.\,\mathbf{Idc}\,(p\,)]\!] = \mathsf{idc}(p\,,\, v_l\,,\, w_l) \quad [\![l\,.\,\mathbf{Rd}\,(p\;\; V)]\!] = \mathsf{rd}(p\,,\, s_l\,,\, [\![V]\!]\,,\, w_l)$$

The compilation of each location state $l\,.\,C$ introduces name c_l and variables s_l, v_l, w_l, l whose visibility from the environment depend on the exported capability recorded in C. Thus, our translation finally introduces the following top-level restrictions: for every location, if no capability have been exported, all these names and variables are restricted; if C has tag **Idu**,

the identifier l is unrestricted; if C has tag **Idc**, the variables w_l and v_l are also unrestricted; if C has tag **Rd**, only the channel c_l is restricted.

Example of transitions in the target language Let us consider how our translation operates on the following transition, which represents player a_1 receiving the result of the game from server e_0 (with $H = h(e_0 + h(s))$).

$$l\,.\,\mathbf{Idu}\,(e_0\;H)\;\mid\;a_1[c_1?(x).P_1]\;\xrightarrow{c_1?l\,.\,\mathbf{Rd}\,(e_0\;s\;\mathbf{11})}\;l\,.\,\mathbf{Rd}\,(e_0\;s\;\mathbf{11})\;\mid\;a_1[P_1\{^{l\,.\,\mathbf{Rd}\,(e_0\;\mathbf{11})}/_x\}]$$

The translated system $\{H/l\} \mid [\![a_1[c_1?(x).P_1]]\!]$ simulates the source transition by an input with label $c_1\,?\,(\,\mathsf{rd}(e_0\,,\,s\,,\,\mathbf{11}\,,\,\mathsf{sign}(\mathsf{h}(s)+\mathsf{h}(s+\mathbf{11})\,,\,\mathsf{sk}(m_{e_0}))\,)$, followed by a series of reductions through the code of parse, including dynamic checks on is_rd and check_idc. In 6 silent steps (including 3 steps for recursive processing of value 11), this yields the process

$$\begin{aligned}&\{H/l\} \mid [\![a_1[P_1]]\!]\{^{\mathsf{rd}(e_0\,,\,s\,,\,\mathbf{11}\,,\,\mathsf{sign}(\mathsf{h}(s)+\mathsf{h}(s+\mathbf{11})\,,\,\mathsf{sk}(m_{e_0})))}/_x\}\\ &\quad\mid \mathsf{repl}\;log!\langle\mathsf{get\_idc}(x)\rangle \mid \nu\, r\,.\,(\mathsf{repl}\;r?(\_).c_1?(x).\mathsf{parse}\;x\;[\![P]\!]_a) \mid R \mid E.\end{aligned}$$

After applying structural equivalence with active substitutions and eliminating the dead loop on channel r, we obtain a system

$$\nu\, s_l\,.\,\nu\, v_l\,.\,\nu\, w_l\,.\,(\{s/s_l\} \mid \varphi(\mathsf{h}(s_l), \mathsf{h}(s_l+\mathbf{11}))_{e_0} \mid [\![a_1[P_1]]\!]\{^{\mathsf{rd}(e_0\,,\,s_l\,,\,\mathbf{11}\,,\,w_l)}/_x\}) \mid R \mid E$$

that matches the translation of the resulting source system above.

3.2.5 Correctness results

The first proposition states that the behaviour of every source system can be simulated by its translation. That is, for any labelled trace of all source systems, there is a labelled trace of the process resulting from its translation. This shows the correctness (or functional adequacy) of our translation. We let $\xrightarrow{\phi}$ (resp. $\xrightarrow{\psi}$) range over series of transitions in the labelled semantics of the source (resp. target) language.

Theorem 3.1 (Functional adequacy)**.** *Let A be a well-formed source system.*
For all series of transitions $A \xrightarrow{\phi}{}^{*} A'$, there exist transitions $[\![A]\!] \xrightarrow{\psi}{}^{*} [\![A']\!]$.

The proof of the theorem is by induction on a series of source transitions between systems in normal forms. For each source transition, we exhibit target transitions that commute with the translation. The full proof can be found in Appendix B.2.2.

The "upwards" direction is more challenging: the trace produced by the translation of a source process A can be related to a trace produced by A *unless* its translation emits the name of a cheating principal on the special channel *bad*. This property uniformly guarantees the security of the translation of all systems with respect to the source semantics, provided that a proof that a principal cheated is a reasonable exceptional outcome for the other principals.

We let $S \longrightarrow_D^* S'$ denote that a target system S goes to S' with a (possibly empty) series of silent deterministic transitions, and let $S \Downarrow M$ abbreviate $S \longrightarrow_D^* \xrightarrow{bad!M} S'$ for some S'; we then say that M is blamed.

Theorem 3.2 (Security)**.** *For all transitions $[\![A]\!] \xrightarrow{\psi}{}^{*} S$ starting from a well-formed source system A, we have*

(1) *either there are source transitions $A \xrightarrow{\phi}{}^{*} A'$ leading to a well-formed source system A' such that $S \longrightarrow_D^* [\![A']\!]$; or $S \Downarrow e$ for some $e \notin \mathcal{A}$;*

(2) *if $S \Downarrow M$, then $M \notin \mathcal{A}$.*

The proof is by induction on the series of transitions in the target language that do not trigger a blame. The first part of the theorem states that either the source semantics is respected, or the implementation at least provides the honest participants with the name of one dishonest principal to blame. Said otherwise, its statement excludes the possibility of cheating without eventual detection. The second part of the theorem expresses that honest participants are never blamed (even in the case some dishonest participants cheat), a necessary property for any optimistic implementation. The full proof can be found in Appendix B.2.3.

The form of our theorem differs from security properties for other programming abstractions (e.g. [Corin et al., 2007, Abadi et al., 2002]), where any run or labelled trace of the cryptographic implementation of a source program is related to a run or labelled trace of the program on the source level. Reflecting a more flexible approach to security, it enables bad runs as long as malicious principals are reliably detected and blamed.

We illustrate how the resolution protocol and the verifications made by the translation of receive suffice to detect write-after-commit attacks. Consider the online game example and suppose that $a_1, a_2 \in \mathcal{A}$ and $e_0 \notin \mathcal{A}$, that is, the server implementation is malicious. In particular, the server implementation may commit location l twice, to convince a_1 that he is the winner with his bid 11 and a_2 that he is the winner with his bid 8. The system composed by the translation of the two clients $[\![A_1 \mid A_2]\!]$ generates a trace

$$[\![A_1 \mid A_2]\!] \to \cdots \to [\![A']\!] \xrightarrow{c_1\,?\,(\,\mathsf{rd}(e_0\,,\,s\,,\,\mathbf{11}\,,\,w)\,)} \xrightarrow{c_2\,?\,(\,\mathsf{rd}(e_0\,,\,s\,,\,8\,,\,w')\,)} S$$

where the seals w and w' sign commitments of l to 11 and 8, respectively.

For the first input transition, there exists a matching source transition, with a resulting source system A'' that includes the location state $l\,.\,\mathbf{Rd}\,(e_0\;s\;\mathbf{11})$. Moreover, the translation of A'' emits the corresponding idc on log.

For the second input transition, however, there is no matching source transition. This would require a capability transition from $\mathbf{Rd}\,(e_0\;s\;\mathbf{11})$ to $\mathbf{Rd}\,(e_0\;s\;8)$, which is excluded by our definition of the $\preceq$ preorder. Instead, the resulting system sends a second **Idc** on log. As soon as the resolution process reads both commitments, it detects that they are inconsistent, and blames e_0 on bad.

Abadi et al. [2002] phrase the security result on abstractions of secure channels and their implementations in terms of behavioural equivalence relations. In our system, an interesting future work is to study if our results can be extended to equivalence relations.

3.3 Offline e-cash

In cryptography, anonymous electronic cash was invented by Chaum [1982]. Also called electronic money or digital cash among others, e-cash is intended to be used in the same situations as the usual cash – to purchase products while keeping one's anonymity. Because the payer's identity must be hidden, both cash and e-cash should be self-contained means of payment: once accepted, the purchase cannot roll back. The main efforts of the bank thus aim to prevent and detect malicious customers to forge cash. While in real life, complex printing and watermarking techniques are used to make cash difficult to reproduce, electronic cash uses cryptography to encode data representing valid coins. Still, any electronic data is a sequence of bytes which can be freely copied, so a malicious customer could try to *double-spend* an electronic coin, that is to use it several times. A usual e-cash scheme includes the following parties:

– a *bank* holds accounts of merchants and clients;
– a *merchant* accepts e-cash from clients and deposits it with its bank;
– a *client* withdraws e-cash from its bank and spends it with merchants.

Banks are the parties who can perform the final check of the validity of a coin, and therefore decide whether to accept it – or blame its client. To get the guarantee of acceptance of a digital

coin by the bank, a merchant can ask an *online* confirmation of the coin validity from the bank before accepting a spend. However to avoid the communication overhead the merchant can accept a user's coin after minimal local checks, accumulate coins and then to deposit them in batches to the bank *offline*. The merchant is then guarnteed that either the coin will be accepted by the bank, or, when this coin has already been deposited by another merchant who received the same coin as payment, the malicious client will be caught and punished. This guarantee should not though compromise the anonymity of well-behaved clients.

Comparison with value commitment Offline e-cash can be seen as a variant of our memory model for value commitment with more interesting secrecy and privacy properties. Coins extend committable cells in the following way:

- the bank who creates the coin is distinct from the client who has the privilege to "commit" the coin to the name of the merchant who should be paid using it;
- the bank does not receive any reference or capability for the created coin;
- there is no way to extract the client's identity from the coin;
- double-spending a coin is the counterpart of multiple commitment of a cell; proving the former requires the evidence that the same coin has been spent twice, while proving the latter requires two capabilities committed to different values.

We design an extension of applied pi that models anonymous offline e-cash by hardwiring its properties in the abstract semantics of the language. To validate the practicality of our abstraction, we relate it to a computational cryptographic scheme [Camenisch et al., 2005]. Indeed, the security goals of the scheme being very complex and technically difficult to achieve, we aim an existing cryptographic implementation rather than a symbolic implementation like the one of Section 3.2. Then, to relate the abstract and the cryptographic layers we need to define an intermediate, abstract cryptographic layer which is syntactically close to the abstract one but which models the principal corruption like in the cryptographic layer. In this intermediate level we encode an equivalent of the *resolution* process to detect and blame cheaters using the logged evidence.

Target cryptographic implementation An important number of implementations of e-cash exist [Chaum, 1982, Chaum et al., 1988, Okamoto and Ohta, 1989, Tsiounis, 1997, Camenisch et al., 2005, 2007a]. We chose to implement our abstract language using the scheme proposed by Camenisch et al. [2005] which guarantees anonymity of clients, detection of fake coins by the bank (also called *balance*) and detection of double-spenders. This work develops complex cryptographic protocols for withdrawal, spending, and depositing e-cash and shows their correctness at the level of computational cryptography. Our abstraction can be used to assess the usability of their cryptographic definitions to reason about the application properties in practice.

Given the properties of some low-level scheme, how far are they from those that a programmer of e-cash applications would expect? What properties does the resulting application offer? With our three-layer system, on one hand a programmer can write short and readable applications using our intuitive primitives, so that one can reason about high-level application properties.On the other hand, there is a considerable gap between the idealized semantics and the cryptographic implementation. Two kinds of discrepancies are to be considered. The intended properties may not be achieved for two kinds of reasons.

- Cryptographic protocols, even if run as intended by honest principals, may fail with negligible probability. In the intermediate layer we introduce primitives that model logic subprotocols of the cryptographic level: bank's and user's side withdrawal, user's and merchant's side spending, merchant's and bank's side deposit. We assume that these primitives always successfully terminate (according to the low-level definitions). We abstract away from interception, replacement, damaging attacks on messages by active adversary,

Figure 3.3: Syntax of the applied pi calculus extended for e-cash

P	::=		$A,\ E$	::=	
	|	$\ldots$		|	$\ldots$
	|	withdraw! $\mathcal{B}\,(x)\,.P$		|	$l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I})$
	|	withdraw? $\mathcal{U}.P$	M	::=	
	|	spend! $\mathcal{B}\,\mathcal{M}\,c\,u$		|	$\ldots$
	|	spend? $\mathcal{B}\ (x)\ (x_1)(x_2).P$		|	$\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)$
	|	deposit! $\mathcal{B}\,M$		|	$Goods$
	|	deposit? $\mathcal{M}\,(x).P$	$\mathcal{I}$	::=	
				|	$\varnothing$
				|	$\{s, \mathcal{I}\}$
				|	$\{\overline{s}, \mathcal{I}\}$

but we still consider that the adversary may have corrupted a subset of principals, so they can for example double-spend.

– Protocols may be run by dishonest principals who misuse them. While code for cheating users may be written in the low and intermediate levels, we exclude any kind of active attack and double-spending from the abstract layer, providing it with a strong defensive semantics.

Results We show that processes in abstract and intermediate level semantics either have the same compliant behaviour or there is a malicious double-spender who can be identified and blamed using reliable evidence of his guilt. The relation to the computational implementation is out of scope of this thesis; the details and proofs of correctness and security for the low level can be found in the online technical report [Adão et al., 2008].

3.3.1 A language for offline e-cash

As a starting point for our formalization of e-cash, we design a toy language suited for intuitive programming of e-cash applications. We extend an instance of applied pi (as presented in Section 2.3) with e-cash primitives and *coins*. The grammar for terms M, processes P, systems A and coin stores $\mathcal{I}$ is given in Figure 3.3.1.

Among names, the principals are denoted with p for a generic role, $\mathcal{U}$ for users, $\mathcal{M}$ for merchants and $\mathcal{B}$ for banks. The set of all principals is statically known and denoted $\mathcal{P}$; the subset of *honest* principals is denoted $\mathcal{H}$.

The set of *coin identifiers* is ranged over by l. Terms M are extended with sale receipts **rcp** (described below) and constants (*Goods*) that model digital goods (for instance .mp3 files).

Coins A *coin* is withdrawn from a bank by a user who can later decide to spend it with one merchant of its choice. The state of each coin in the system is modeled by an extended process $l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I})$ where l is the unique coin identifier, b is the identity of the bank that issued the coin, u is the principal that withdrew the coin, and $\mathcal{I}$ is the list of the transactions in which this coin was used for payment. The user identity u is only recorded to maintain consistency and well-formedness of configurations, and cannot be extracted from the coin. We explain further that the semantics of our high-level language disallows multiple spendings, so the list is at most of length 1. However since we reuse the syntax for the intermediate level language where multiple spendings are tolerated, we keep the general notation for lists with arbitrary number of transactions. A transaction is denoted with its identifier. During its lifetime a coin goes through the following states (listed in their chronological order):

– $l.(\mathcal{B}\,\mathcal{U}\,\varnothing)$ denotes a coin that has been withdrawn but not yet spent or deposited;
– $l.(\mathcal{B}\,\mathcal{U}\,\{\,s\,\})$ denotes the same coin once it has been spent in the session s;
– $l.(\mathcal{B}\,\mathcal{U}\,\{\,\overline{s}\,\})$ denotes the same coin once it has been deposited.

Figure 3.4: Reduction rules for e-cash

(Withdraw)

$$\frac{n \notin fn(P_1) \cup fn(P_2)}{\mathcal{U}[\mathsf{withdraw!}\,\mathcal{B}\,(x)\,.P_1] \mid \mathcal{B}[\mathsf{withdraw?}\,\mathcal{U}.P_2] \rightarrow_a \nu\, l\,.\,(\mathcal{U}[P_1\{^{l}/_{x}\}] \mid \mathcal{B}[P_2] \mid l.(\mathcal{B}\,\mathcal{U}\,\varnothing))}$$

(Spend)

$$\frac{(\,s \notin fn(P)\,)}{\mathcal{U}[\mathsf{spend!}\,\mathcal{B}\,\mathcal{M}\,c\,l] \mid \mathcal{M}[\mathsf{spend?}\,\mathcal{B}\,(x)\,(x_1)(x_2).P] \mid l.(\mathcal{B}\,\mathcal{U}\,\varnothing) \rightarrow_a \nu\, s\,.\,(\mathcal{M}[P\{^{c}/_{x}\}\{^{\langle l,s\rangle}/_{x_1}\}\{^{\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)}/_{x_2}\}] \mid l.(\mathcal{B}\,\mathcal{U}\,\{\,s\,\}))}$$

(Deposit)

$$\frac{}{\mathcal{M}[\mathsf{deposit!}\,\mathcal{B}\,\langle l,s\rangle] \mid \mathcal{B}[\mathsf{deposit?}\,\mathcal{M}\,(x).P] \mid l.(\mathcal{B}\,\mathcal{U}\,\{\,s\,\}) \rightarrow_a \mathcal{B}[P\{^{\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)}/_{x}\}] \mid l.(\mathcal{B}\,\mathcal{U}\,\{\,\overline{s}\,\})}$$

The scheme developed by Camenisch et al. [2005] allows to users to withdraw *wallets* of coins. We only consider wallets that contain a single coin; with a slight abuse of terminology we refer to "wallets" as "coins".

E-cash primitives We introduce three pairs of processes that manipulate coins; these actions represent the intent of each participant to run one of the three e-cash subprotocols (withdraw, spend, or deposit) with a remote peer. For each protocol, there is an initiator and a responder, whose actions end with ! and ?, respectively. By convention, the first argument of these actions (except for spend?, waiting an anonymous payment) indicates the intended partner.

Process $\mathsf{withdraw!}\,\mathcal{B}\,(x)\,.P$ allows to initiate the withdraw protocol with the principal $\mathcal{B}$ and continue with process P where x is bound to the identifier of the withdrawn coin. Process $\mathsf{withdraw?}\,\mathcal{U}.P$ waits for the principal $\mathcal{U}$ to initiate the withdraw protocol and continues as process P. Process $\mathsf{spend!}\,\mathcal{B}\,\mathcal{M}\,c\,l$ allows to spend a coin identified as l via the bank $\mathcal{B}$ with the merchant $\mathcal{M}$ for the goods identified with c. Since spending involves some kind of exchange, the correlator c provided by the user allow to specify details on the purchased digital good (see further discussion of the language design). For instance, c may be the name of the purchased song and the information on the delivery. The spend! process has no continuation so that the end of a particular spend transaction cannot be detected and the anonymity of spendings is preserved. Process $\mathsf{spend?}\,\mathcal{B}\,(x)\,(y)(z).P$ waits for some principal to spend a coin via bank $\mathcal{B}$ and continues as process P where x is bound to the product information, y is bound to the pair (coin identifier, session identifier), and z is bound to the *receipt*. Process $\mathsf{deposit!}\,\mathcal{B}\,\langle l,s\rangle$ allows to deposit a coin l spent during transaction s to the bank $\mathcal{B}$; like spend!, deposit! is asynchronous. Process $\mathsf{deposit?}\,\mathcal{M}\,(x).P$ waits for the principal $\mathcal{M}$ to deposit a coin and continues as process P.

Receipts After accepting a payment, the merchant obtains a transferable *receipt* $\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)$ which witnesses spending of the coin l issued by the bank $\mathcal{B}$ to the user $\mathcal{U}$ with the merchant $\mathcal{M}$, recorded with the fresh session identifier s. Sale receipts can be communicated over channels; they can also be manipulated using special functions, according to the equational theory below.

$$\mathbf{bank}\,(\,\mathbf{rcp}(n,\mathcal{B},\mathcal{U},s)\,) = \mathcal{B}$$
$$\mathbf{merchant}\,(\,\mathbf{rcp}(n,\mathcal{B},\mathcal{U},s)\,) = s$$

The **merchant** and **bank** functions yield the merchant's and the bank's name from sale receipts, respectively. Receipts will be used by our more realistic, intermediate semantics, to detect and identify double-spenders.

Reduction semantics Let $A \rightarrow_a A'$ denote the relation "Configuration A reduces to configuration A'", and let $A \Rightarrow A'$ denote its transitive and reflexive closure. We extend the reduction semantics of applied pi calculus with three rules for e-cash transactions given in Figure 3.3.1.

Rule WITHDRAW describes synchronization between a user and a bank willing to run the Withdraw protocol with each other. The bank supposedly holds this user's account and the

account has a sufficient balance. In the resulting configuration a fresh coin is created and its identifier l is bound in the user's continuation while the bank does not get access to the coin. The coin state records that the created coin has not yet been spent (the store is empty).

Rule SPEND describes the interaction between a user in possession of a coin identifier l corresponding to a non-spent coin with a merchant willing to run the spend protocol with each other. Both the user and the merchant must have their accounts in the same bank. In the resulting configuration a unique fresh transaction identifier s is generated; then the session correlator c, the coin identifier paired with the transaction identifier, $\langle l,s \rangle$, as well as the receipt $\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)$ are bound in the continuation process of the merchant. The coin state records the transaction identifier s in its store; thus the same rule cannot be applied another time. (Note that the user name is only included in the coin and receipt only for consistency but it is not visible to the attacker.)

Rule DEPOSIT describes the interaction between a merchant and its bank who are willing to run the Deposit protocol with each other. The merchant deposits the coin by providing its identifier as well as the identifier of the transaction through which he got the coin. In the resulting configuration the bank obtains the receipt $\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)$ and the coin finally is marked as being in final state; we denote this by overlining the transaction identifier s in the coin store $(l.(\mathcal{B}\,\mathcal{U}\,\{\,\overline{s}\,\}))$. As for the SPEND rule, the coin state is record and thus the same rule cannot be applied another time. The bank supposedly credits the merchant's account after successful application of this rule.

Well-formed configurations Configurations, or systems, denoted A, are distributed parallel compositions of local processes run by at most a fixed subset of honest principals, written $\mathcal{H}_A$. Configurations also include the global state for each known coin currently in circulation.

Up to structural equivalence, all configurations are of the form

$$\nu\mathcal{N}_b.\ \prod_{p\in\mathcal{H}_A} p[P_p] \mid \prod_{l\in\mathcal{N}_c} l.C_l$$

where $\mathcal{N}_b$ are the secret names of the configuration, and $\mathcal{N}_c$ are the coin identifiers. This notation allows for example to refer to the coins that have been issued by honest banks to honest users but not yet spent: $\mathcal{N}_c \cap \mathcal{N}_b$.

Definition 3.3.1. An *initial* configuration, denoted A_0, is a configuration that contains no coins and no receipts. A *well-formed* configuration is a configuration where for any coin identifier $l \in \mathcal{N}_c$, there is *at most* one coin, and l may only occur

- in a spending process $\mathsf{spend!}\,\mathcal{M}\,\mathcal{B}\,c\,l$ when $C_l = l.(\mathcal{B}\,\mathcal{U}\,\varnothing)$ and $(l \in \mathcal{N}_b$ if $\mathcal{B} \in \mathcal{H})$.
- in a depositing process $\mathsf{deposit!}\,\mathcal{B}\,\langle l,s\rangle$ when $s \in \mathcal{N}_b$ and $C_l = l.(\mathcal{B}\,\mathcal{U}\,\{\,s\,\})$.
- in a receipt $\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)$ when $C_l = l.(\mathcal{B}\,\mathcal{U}\,\{\,s\,\})$ or $C_l = l.(\mathcal{B}\,\mathcal{U}\,\{\,\overline{s}\,\})$.

Trivially, initial configurations are well-formed. Our reduction semantics guarantees that each coin can only be spent once: for any $l \in \mathcal{N}_c$, if $C_l = l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I})$ then $|\mathcal{I}|$ is at most equal to 1.

Lemma 3.3 (Preservation)**.** *Well-formed high level configurations are stable by reductions. For any well-formed high level A, if there is a high level system A' such that $A \Rightarrow A'$ then A' is well-formed.*

3.3.2 Properties of the language

The e-cash protocols that we model [Camenisch et al., 2005] enjoy several security properties, including correctness, balance and anonymity. Formally, they are all example properties following from our language semantics.

We say that configurations A and A' are contextually equivalent, denoted $A \approx A'$, when for all context C and channel name c, we have that $C[A]$ can output on c iff $C[A']$ can output on c.

– *Correctness*
  An honest user can withdraw from an honest bank, according to rule WITHDRAW.
  An honest user can spend with an honest merchant according to rule SPEND. Whenever an honest merchant obtains a pair of coin and session identifiers through a spend (this is the only way to obtain those, in our semantics), this merchant will be able to deposit at the bank according to rule DEPOSIT.
– *Balance*
  For any series of reductions of an initial high level configuration, rule DEPOSIT cannot be applied more often than rule WITHDRAW for each coin. First, we have that deposit? must interact with deposit parameterized with the coin and transaction identifiers which are generated when spend? interacts with spend itself parameterized by the coin identifier which in turn is generated after the withdrawal. Second, both spend and deposit can only occur once for every coin. So that each deposit corresponds exactly to one withdrawal.
– *Anonymity of users.* A bank cannot learn anything about a user's spendings. Indeed, executing withdraw? does not provide any information on the coin. The deposit process only provides a blinded receipt $\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)$ for which no high level equations/rules allow to extract the user's identity.
  For instance, the following systems are contextually equivalent (merchant $\mathcal{M}$ cannot distinguish payments from p_1 and p_2, if they come with the same correlator):

$$p_1[\mathsf{spend!}\,\mathcal{B}\,\mathcal{M}\,c\,l] \approx p_2[\mathsf{spend!}\,\mathcal{B}\,\mathcal{M}\,c\,l]$$

Discussion of our language design Our language captures the essence of e-cash protocols and puts forward the base properties which are not obvious at all in their low level implementation. However, as it often happens, this model is too pure to conveniently express real world applications.

Our current design choices were driven by the implementation of Camenisch et al. [2005]. Our model can well express anonymous money donations. But the language is not rich enough to express a realistic electronic commerce of digital goods. In real life there is a need of a frame for sale negotiation, without losing anonymity of clients. To tackle this, our sale primitives include a primitive payload handling. A client passes a correlator to the merchant when spending. It is up to the client to ensure that the correlator does not disclose his identity. In practice it may be a nonce with which the merchant can label the goods when posting them on some public channel (and provided that the others readers of the channel are trusted). Or it can be a password for some independent goods storage service, where the merchant sends the goods and the client uses the password to get hold of them. We may instead assume that every merchant only sells one type of products, all participants have access to the description and availability information. Also in practice our design is not fair to the users. A user who spent a coin does not get any receipt, and so if the merchant is malicious he can even lose his money.

Example 3.3.1 (Tagged delivery)**.** *A client can provide a tag k to be attached to the purchased electronic good at delivery. Let c be a public channel used for delivery. On receiving his puchase, the client compares its tag with the one he provided.*

$$\mathcal{U}[\mathsf{withdraw!}\,\mathcal{B}\,(x)\,.\nu\,k\,.\,(\mathsf{spend!}\,\mathcal{B}\,\mathcal{M}\,r\,x \mid c?(y).\mathsf{if}\ \mathbf{snd}\,(\,y\,) = k\ \mathsf{then}\ P_{\mathcal{U}}\ \mathsf{else}\ 0)]$$

The merchant attaches the tag chosen by the client (here bound to variable x) to the purchase and sends it on channel c.

$$\mathcal{M}_1[\mathsf{spend?}\,\mathcal{B}\ (x)\ (y)(z).(\mathsf{deposit!}\,\mathcal{B}\,y \mid c\,!\,\langle \mathbf{Song.mp3}, x\rangle \mid P_{\mathcal{M}_1})]$$

3.3.3 Log-based implementation

As a further step to modelling realistic offline e-cash we propose an *intermediate level semantics* that allows double spending. We also propose a fraud detection mechanism based on logging.

Coins, revisited We extend the semantics of coins to account for double-spending (and we use the same syntax as before, shown in Figure 3.3.1.

A coin contains the list of the sale identifiers it has been used for. We recall that the abstract semantics only allows this list to be empty, or a singleton.

Information received during an e-cash protocol on a coin is sufficient to construct or update the coin state in the system.We express compliant updates for coins as a preorder on coins.

We formalize the notion of "compatible coin" by defining an irreflexive preorder $\prec$ on coins. Intuitively, $l.C \prec l.C'$ holds if the information in C can be derived from that in C'. The special marker $\perp$ represents the absence of knowledge on a coin. The order over coins is defined by the axioms below:

$$\begin{array}{ll} l.\perp \prec l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}) & \text{when } \mathcal{B} \notin \mathcal{H} \\ l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}) \prec l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}') & \text{when } \quad \mathcal{I} \prec \mathcal{I}' \end{array}$$

The only particular constraint on updates is that only coins owned by adversarial banks can be created by the environment. All other operations are allowed to happen in the environment, and the honest configuration can only learn them after the fact, because a logically following protocol is run, or because a corresponding receipt has been received. The order over coin contents is the following:

$$\varnothing \prec \{\, s \,\} \prec \{\, \overline{s} \,\}$$

Compared to the preorder on capabilities in Section 3.2, the updates for coins have two flavours. First, since "commitments" (spends and deposits made by the environment) are distributed in e-cash, we need the irreflexivity of the preorder to ensure that the update for the corresponding protocols can only be done once. We write $C \uplus C'$ for the sup of C and C' with respect to $\prec$, when it exists (for better visibility, we often mark it in red: $\uplus$).

Second, the receipts are quite similar to the marshallable capabilities for committable cells, a same receipt may be received several times. We write $C \curlyvee C'$ for the sup of C and C' with respect to the reflexive closure of $\prec$, when it exists.

We rewrite the abstract reduction semantics into the following equivalent form:

$$\frac{(\, l \notin fn(P_1) \,\wedge\, l \notin fn(P_2)\,)}{\begin{array}{c}\mathcal{U}[\mathsf{withdraw!}\,\mathcal{B}\,(x)\,.P_1] \mid \mathcal{B}[\mathsf{withdraw?}\,\mathcal{U}.P_2] \to_i \\ \nu\, l\,.\,(\mathcal{U}[P_1\{{}^{l}/_{x}\}] \mid \mathcal{B}[P_2] \mid l.(\mathcal{B}\,\mathcal{U}\,\varnothing))\end{array}}\ \text{(WithdrawSup)}$$

$$\frac{s \notin fn(P)}{\begin{array}{c}\mathcal{U}[\mathsf{spend!}\,\mathcal{B}\,\mathcal{M}\,c\,l] \mid \mathcal{M}[\mathsf{spend?}\,\mathcal{B}\,(x)\,(x_1)(x_2).P] \mid l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}) \to_i \\ \nu\, s\,.\,(\mathcal{M}[P\{{}^{c}/_{x}\}\{{}^{\langle l,s\rangle}/_{x_1}\}\{{}^{\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)}/_{x_2}\}] \mid l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}\uplus\{\, s \,\}))\end{array}}\ \text{(SpendSup)}$$

$$\frac{}{\begin{array}{c}\mathcal{M}[\mathsf{deposit!}\,\mathcal{B}\,\langle l,s\rangle] \mid \mathcal{B}[\mathsf{deposit?}\,\mathcal{M}\,(x).P] \mid l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}) \to_i \\ \mathcal{B}[P\{{}^{\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)}/_{x}\}] \mid l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}\uplus\{\, \overline{s} \,\})\end{array}}\ \text{(DepositSup)}$$

We equate $l.\perp|A$ to A, so that the input rule covers the case of an input carrying fresh, unknown coins from the environment. Like for the committed cells, the resulting configuration must be well-formed, which excludes the introduction of a fresh coin state for l if one already exists in the system.

Intermediate level semantics A coin is now only used for

- recording valid coins issued by honest banks, so that an adversary cannot forge them;
- recording deposits, so that an adversary cannot deposit exactly the same coin several times (replay attacks by a merchant).

Since multiple spendings are allowed, the list of sale identifiers for which the coin has been spent or deposited can have size greater than 1.

We have the following invariant for a coin transaction set $\mathcal{I}$: a transaction number s either (a) unknown, then $s \notin \mathcal{I}$; (b) or spent, then $s \in \mathcal{I}$ or $\overline{s} \in \mathcal{I}$; (c) or deposited, then $\overline{s} \in \mathcal{I}$.

In the intermediate semantics, the symbols $\curlyvee$ and $\uplus$ applied to a list of sale identifiers yield this list where the corresponding sale identifier, if any, is updated to maintain the invariant, or is added to the list.

The reduction rules for the intermediate level are the same as for the abstract semantics, except for the spend rule which we replace with the following rule We keep the same reduction rules for the intermediate level, (in the coin update the strict sup $\uplus$ is replaced with $\curlyvee$):

(SpendInt)

$$\frac{s \notin fn(P)}{\mathcal{U}[\mathsf{spend!}\,\mathcal{B}\,\mathcal{M}\,c\,l] \mid \mathcal{M}[\mathsf{spend?}\,\mathcal{B}\,(x)\,(x_1)(x_2).P] \mid l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}) \rightarrow_i \nu\, s\,.\,(\mathcal{M}[P\{^{c}/_{x}\}\{^{\langle l,s\rangle}/_{x_1}\}\{^{\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)}/_{x_2}\}] \mid l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}\curlyvee\{\,s\,\}))}$$

With the "set" semantics of $\uplus$, during deposit if the sale has already been recorded for the coin, then the deposit is not accepted since the merchant is trying to deposit twice the same coin.

The syntax of the intermediate language processes is the same; additionally we provide a special process

$$\mathsf{identify}\ M\ M'\ (x).P$$

We allow participants to collect evidence (the rcp receipts for coins) and to detect and identify double-spenders, using identify. When given two receipts for the same coin but with different transaction numbers – characteristic for a double-spent coin – it reduces to P where x is bound to the extracted identity of the coin's spender. This is the only case when the system reveals the user identity. Otherwise – the evidence is either insufficient for blaming or invalid –,the process is stuck. We thus guarantee *weak exculpability*: a user can only be successfully blamed if he indeed double-spent.

(Identify)

$$\frac{s \neq s'}{p[\mathsf{identify}\ \mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)\ \mathbf{rcp}(l,\mathcal{B},\mathcal{U},s')\ (x).P] \longrightarrow_i p[P\{^{\mathcal{U}}/_{x}\}]}$$

To detect double spending in practice, one needs to collect all the receipts for each coin. We do not hard-wire such logging but only ensure that there is enough data available to record in logs – the ready-to-log receipts are available when accepting a spend or a deposit. Typically, logs should be kept on bank's side but we leave the choice of their efficient implementation to the programmer.

Systematic logging and fraud detection To achieve a strong guarantee of *a posteriori* fraud detection, identify process should be introduced systematically into processes running at the intermediate level, at all possible points of fraud.

Similarly to the optimistic implementation of value commitment in Section 3.2, we introduce

logging and a global resolution process. The implementation is defined by the following clauses:

$$\begin{aligned}
[\![\mathsf{spend?}\ p\ (x)\ (y)(z).P]\!] &= \mathsf{spend?}\ p\ (x)\ (y)(z).(P \mid \mathsf{repl}\ \mathit{log}\,!\,z)\\
[\![\mathsf{deposit?}\ p\ (x).P]\!] &= \mathsf{deposit?}\ p\ (x).(P \mid \mathsf{repl}\ \mathit{log}\,!\,x)\\
[\![u?(x).P]\!] &= u?(x).(P \mid \mathsf{if}\ \mathbf{isrcp}((\ x\ )) = \mathbf{ok}\ \mathsf{then}\ \mathsf{repl}\ \mathit{log}\,!\,x\ \mathsf{else}\ 0)\\
[\![p[P]]\!] &= p[[\![P]\!]]\\
\mathit{Resolution} &= \mathsf{repl}\ \mathit{log}?(x_1).\mathit{log}?(x_2).\mathsf{identify}\ x_1\ x_2\ (x).\mathit{bad}\,!\,x\\
[\![A]\!] &= [\![A]\!] \mid \mathit{Resolution}
\end{aligned}$$

The translation of all other constructs is homomorphic. When accepting a spend, a deposit, or receiving a sale receipt via a normal channel the corresponding receipts are recorded over channel log. The replicated contents of this channel models the system log. We deploy the *Resolution* process on top of the translation of systems. The *Resolution* process repeatedly reads and identifies pairs of **rcp** receipts. When a fraud is exposed by the receipts, the process identify reduces and the identity of the guilty user is sent over channel *bad*. We say that a system *detects cheating* if it outputs on channel *bad*.

For our theorems we only need to compile initial high level systems. Conversely, their translations after reductions and discarding the history, may be decompiled into well-formed high level systems.

3.3.4 Model and translation of environment interactions

Our results relate the behaviour of abstract systems and their intermediate level implementations. We define a labelled semantics that precisely characterizes the compliant interactions between a system of honest principals and an arbitrary, possibly hostile environment.

The labels have the following shape: action, subject, peer, extra arguments. For the rules with label $\phi\ p_1\ p_2\ \ldots$ we systematically have the side condition $p_1 \in \mathcal{H}$ and $p_2 \notin \mathcal{H}$. Labels $\nu n.\phi!\ldots$ model standard scope extrusion: a secret name n is sent to the adversary. Labels $\nu n.\phi?\ldots$ model "intrusion": learning the fresh name n from the adversary (with the guarantee that n does not clash with any of the free names of the configuration).

We require that the following LTS rules are only applied to well-formed systems, and that they produce well-formed systems. We present the other labelled transitions from the point of view of different honest users who interact with the environment: honest banks, honest users and honest merchants. Internal reductions between honest users appear as silent transitions to the adversary.

Abstract layer An honest user can withdraw money from a bank and spend it with some merchant. Both can happen either with an honest or a dishonest peer: the former interaction is described by a reduction rule, the second by a labelled transition. At withdrawal, a coin is created; at spend, a fresh session number is generated at the adversary's side, so the coin state is updated with a fresh identifier.

$$\mathcal{U}[\mathsf{withdraw!}\,\mathcal{B}\,(x)\,.P] \xrightarrow{\mathsf{withdraw!}\,\mathcal{U}\,\mathcal{B}}_a \nu\, l\,.\,(\mathcal{U}[P\{{}^{l}/_{x}\}] \mid l.(\mathcal{B}\,\mathcal{U}\,\varnothing))\,(\,l \notin fn(P)\,)$$

$$\mathcal{U}[\mathsf{spend!}\,\mathcal{B}\,\mathcal{M}\,c\,l] \mid l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}) \xrightarrow{\nu s.\mathsf{spend!}\,\mathcal{M}\,\mathcal{B}\,l\,s}_a l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}\uplus\{\,s\,\})$$

$$\mathcal{B}[\mathsf{withdraw?}\,\mathcal{U}.P] \xrightarrow{\nu l.\mathsf{withdraw?}\,\mathcal{B}\,\mathcal{U}}_a \mathcal{B}[P] \mid l.(\mathcal{B}\,\mathcal{U}\,\varnothing)$$

$$\mathcal{B}[\mathsf{deposit?}\,\mathcal{M}\,(x).P] \mid l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}) \xrightarrow{\mathsf{deposit?}\,\mathcal{B}\,\mathcal{M}\,l\,s\,\mathcal{U}}_a \mathcal{B}[P\{^{\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)}/_{x}\}] \mid l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}\uplus\{\,\overline{s}\,\})$$

$$\mathcal{M}[\mathsf{spend?}\,\mathcal{B}\ (x)\ (x_1)(x_2).P] \mid l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}) \xrightarrow{\nu s.\mathsf{spend?}\,\mathcal{M}\,\mathcal{B}\,c\,l\,s\,\mathcal{U}}_a \mathcal{M}[P\{{}^{c}/_{x}\}\{{}^{\langle l,s\rangle}/_{x_1}\}\{^{\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)}/_{x_2}\}] \mid (l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}\uplus\{\,s\,\}))$$

$$\mathcal{M}[\mathsf{deposit!}\,\mathcal{B}\,\langle l,s\rangle] \mid l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}) \xrightarrow{\mathsf{deposit!}\,\mathcal{M}\,\mathcal{B}\,l\,s}_a l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}\uplus\{\,\overline{s}\,\})$$

$$\mathcal{A}[c?(x).P] \mid l.C_l \xrightarrow{c\,?\,\mathbf{rcp}(l,\mathcal{BU},s)}_a \mathcal{A}[P\{^{\mathbf{rcp}(l,\mathcal{BU},s)}/_x\}] \mid (l.C_l \curlyvee l.(\mathcal{BU}\{\,s\,\}))$$

To achieve the Balance property, an honest bank needs to tackle two issues, and this is where the global coin states come into play.

- The adversary may try to fake a coin; we record all valid coins l at their withdrawal as $l.(\mathcal{BU}\,\varnothing)$. A deposit of a coin is accepted only if the corresponding coin exists – it acts as a proof of authenticity. The coin may have been already spent or still be empty.
  For incompatible updates, the sup does not exist, so the rule does not apply.
- the adversary may try to spend the same coin several times.
  The received transaction number must be either the same as already recorded by the coin (if the spender is honest) or fresh. After deposit the coin goes to its final state and cannot be spent or deposited any more.

When accepting a spend, an honest merchant must make sure that if the coin's issuing bank is honest then a corresponding empty coin is known to the system, otherwise a fresh coin is created. A fresh transaction number is recorded in the coin in both cases. At deposit, the coin goes to the final state.

Most of rules can be rewritten in a simpler form, by inlining the constraints on coins, for instance the rule for deposit input can be rewritten as $\mathcal{M}[\mathsf{deposit!}\,\mathcal{B}\,\langle l,s\rangle] \mid l.(\mathcal{BU}\{\,s\,\}) \xrightarrow{\mathsf{deposit!}\,\mathcal{M}\,\mathcal{B}\,l\,s}_a l.(\mathcal{BU}\{\,\overline{s}\,\})$. However we keep the constraints on coins separate so that we can take advantage of the modularity of the rules to define the semantics of the intermediate level.

We have standard rules for sending terms, scope extrusion and receiving principal names.

As a sanity check, we verify that interactions with an abstract environment, represented by a series of transitions, can also occur using reductions within an evaluation context.

Intermediate layer In the intermediate layer, the rules for spend input and output are relaxed: instead of the strict preorder sup $\uplus$, the coin is updated with the reflexive closure $\curlyvee$.

Synchronous versus asynchronous LTS We have a synchronous labelled semantics: for example, an adversary can only spend a coin when the corresponding honest merchant is expecting a spend. Since the corresponding spend action has no continuation, contexts are weaker than labels. We can change our semantics into an asynchronous one by splitting spend and deposit rules into two each: first the initiating message is buffered on the receiver's side, and then it is silently consumed by the receiver. assuming that the receiver of an asynchronous message sends back to the expeditor an acknowledgement: this only happens for spends and deposits, and it is reasonable to assume that merchants and banks are always willing to accept.

Our correctness results are phrased using trace equivalences, defined below.

Definition 3.3.2 (Trace Equivalence)**.** Two systems are (weakly) *trace-equivalent* when their labelled transitions have the same series of non-silent labels up to renaming.

Definition 3.3.3 (Observational Determinism)**.** A system A is *deterministic* when, for all transitions with the same non-silent labels $A \xrightarrow{\phi} A_1$ and $A \xrightarrow{\phi} A_2$, the two systems A_1 and A_2 are trace-equivalent.

Lemma 3.4. *Observational determinism is preserved by transitions.*

Proof: Suppose that A is deterministic, $A \xrightarrow{\phi} A'$, and A' is not deterministic. Then, there are ψ, A'_1, A'_2 such that $A' \xrightarrow{\psi} A'_1$, $A' \xrightarrow{\psi} A'_2$ and A'_1 and A'_2 are not trace-equivalent. Then $A \xrightarrow{\phi\psi} A'_1$ and $A \xrightarrow{\phi\psi} A'_2$ but A'_1 and A'_2 are not trace-equivalent which contradicts the determinism of A. □

3.3.5 Correctness results

We obtain an "optimistic security" result for e-cash and its log-based implementation. We restrict configurations to observationally deterministic systems to avoid giving too much power to the adversary.

We use the following notation: we write $A \xrightarrow{\phi}_a$ for $\exists A'$, such that $A \xrightarrow{\phi}_a A'$ when the shape of A' is irrelevant for what follows. Let $A \sim A'$ (resp., $A \sim_i A'$) denote weak trace equivalence for high (respectively, intermediate) level systems.

First, we formally define observable fraud – *double-accept* – as receiving evidence of double spending. *Honest equivalence* is then defined as equivalence of systems modulo double-accepting.

Definition 3.3.4 (Double-accept and Honest Equivalence)**.** Trace $A \xrightarrow{\phi}_i$ *accepts a coin* l for sale s, bank $\mathcal{B}$, merchant $\mathcal{M}$, and user $\mathcal{U}$ when $\mathsf{spend?}\,\mathcal{M}\,\mathcal{B}\,\_\,l\,s\,\mathcal{U} \in \phi$, or $\mathsf{deposit?}\,\mathcal{B}\,\mathcal{M}\,l\,s\,\mathcal{U} \in \phi$, or $\_?\,(\,\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)\,) \in \phi$.

Trace $A \xrightarrow{\phi}_i$ *double-accepts* if for some $l, s, s', \mathcal{B}, \mathcal{M}, \mathcal{U}$, ϕ accepts l for sale s and ϕ accepts l for sale s' and the same principals $\mathcal{B}, \mathcal{M}, \mathcal{U}$. Two systems are *honestly equivalent*, denoted $A \sim_w A'$, whenever for all ϕ such that $A \xrightarrow{\phi}_i$ does not double-accept, $A \xrightarrow{\phi}_i$ iff $A' \xrightarrow{\phi}_i$.

Lemma 3.5 (Trace lifting)**.** *For any initial high-level system* A_0*, if* $A_0 \xrightarrow{\phi}_i$ *does not witness cheating, then* $A_0 \xrightarrow{\phi}_a$.

The proof is by induction on the intermediate level trace ϕ. We use the fact that high-level systems are a syntactic subset of intermediate systems.

We have a variant of optimistic security. On one hand we show that the intermediate semantics simulates the high level semantics, modulo double-accepting (Theorem 3.6). On the other hand, we show that when wrapped with our logged-based implementation, systems eventually detect double-spending and identify and blame the fraudulent user (Theorem 3.7).

Theorem 3.6 (Optimistic equivalence)**.** *Let* A_0 *and* A'_0 *be two initial high-level systems. We have*

(1) $A \xrightarrow{\phi}_a$ *iff* $A \xrightarrow{\phi}_i$ *does not double-accept;*

(2) $A_0 \sim A'_0$ *iff* $A_0 \sim_w A'_0$*;*

(3) *if* $A_0 \sim_i A'_0$ *then* $A_0 \sim_w A'_0$.

Proof: (1) By induction on the abstract trace ϕ.

(2) ($\Rightarrow$) Suppose that $A_0 \xrightarrow{\phi}_i$ does not witness cheating. Then $A_0 \xrightarrow{\phi}_a$ by Lemma 3.5, $A'_0 \xrightarrow{\phi}_a$ by hypothesis, and $A'_0 \xrightarrow{\phi}_i$ by (1). Finally, by symmetry, we get $A_0 \sim_w A'_0$.
($\Leftarrow$) Suppose that $A_0 \xrightarrow{\phi}_a$. Then $A_0 \xrightarrow{\phi}_i$ does not witness cheating by (1), $A'_0 \xrightarrow{\phi}_i$ by hypothesis, and $A'_0 \xrightarrow{\phi}_a$ by Lemma 3.5. Finally, by symmetry, we get $A_0 \sim A'_0$.

(3) $\sim_w$ is a restriction of $\sim_i$ to a smaller set of traces.

□

Theorem 3.7 (Detectability)**.** *Let* A *be a high level system. If when run using the intermediate semantics* $A \xrightarrow{\phi}_i$ *double-accepts then its translation* $[\![A]\!]$ *detects cheating.*

The proof sketch can be found in Appendix C.2.

3.4 Related work on the use of audit logs

We give a short overview of other optimistic protocols, conjecture the properties of the logs they are using, and discuss the existing implementation techniques for secure logs.

Fair exchange and non-repudiation protocols [Kremer et al., 2002] Fair non-repudiation protocols must ensure that when Alice exchanges some data to Bob, either both Alice and Bob can deny their participation in this communication, or both of them possess irrefutable evidence of the other party's participation. Ideally, Alice exchanges the message and the non-repudiation of origin evidence against Bob's non-repudiation of receipt evidence. The closest to what we call an optimistic protocol are the protocols with an *offline* trusted third party (TTP). With this approach participants only contact the TTP in case of cheating or network problem, so that it reconstructs the missing evidence from the partial evidence available so far, or invalidates the session.

Electronic voting Protocols for electronic voting are fairly complex and by their nature require a form of trust from the voters; this requirement is often unrealistic. To this end most of the voting protocols use optimistic subprotocols, for instance to generate the initial cryptographic material for an election (the tallies).

Cut and choose principle in cryptography is inspired by the general fair division method "I cut, you choose". If Alice must use a secret piece of data which Bob must trust, she can generate n fresh secrets, encrypt them and let Bob choose $n-1$ blindly. Then she discloses these $n-1$ secrets so that Bob can check that Alice did not cheat on these data. Then Bob assumes that the last secret which Alice keeps encrypted to reuse it for later protocol needs is valid and trustworthy. Otherwise Bob could have discovered the cheat by choosing it for verification, with a big probability. All the n secrets must be committed to by Alice and revealed on Bob's demand. Thus Alice cannot replace values of Bob's choice to hide her cheat.

3.4.1 Online games

Mental poker Mental poker [Shamir et al., 1981] allows playing a fair game between physically distant players without involving a trusted third party, it is particularly used in gambling over the Internet. In electronic poker,or e-poker, for example, it is very difficult to guarantee that the card drawn by each player will remain secret for others and honest, that the card deck will be fairly random, and that each party has enough evidence to resolve a dispute that may arise.

In the protocol proposed in Castellà-Roca et al. [2003] for e-poker, to achieve randomness each player generates a random permutation of card and commits to it using commitment scheme. The deck is composed from all players' permutations.

The players' reversed cards are encrypted during the game in such a way that the permutations commute with encryption. Opening a card corresponds to its decryption. Early commitment of the players' permutations is important at the end of the game. Players then reveal their encryption keys and their permutations for auditing.

NVE The optimistic approach with audit trails has been applied to networked virtual environments (NVE) by Jha et al. [2007]. Cheating in virtual reality games may include compromising the game rules or changing the order of events after the fact. An NVE is composed by a server which maintains the global game state, and clients who possess a partial view of the game. Due to limited computational and bandwidth capacities of the server, it only maintains the abstract global state, the concrete states are computed by the clients and the resulting abstract updates are sent back to the server. Maintaining the consistence of the global game state is a challenging task that the authors achieve with auditing. Along with the normal exchanges with the NVE Server, each client regularly sends its hashed and timestamped concrete state to the trusted Audit Server. The latter may initialize the audit procedure from one of these messages, asking the client to send the whole concrete states corresponding to the hashes in order to make

compliance verifications. Thus the semantic integrity of the NVE can be checked with a small network overhead.

3.4.2 Multi-party protocols

Lockstep protocol has similarities with the multi-party protocol presented in Section 3.2.2. Baughman and Levine [2001] study cheat prevention in decentralized multi-player games and allow optimistic (vs conservative) event processing without rolling back. They identify several kinds of attacks and their solutions comparable to those we are using for committed cells.

(1) Lookahead cheat: in the same round, a cheating player may wait to see the decisions of all other players. A known solution is *lockstep synchronization*. All players first send their commitments, and only then plaintexts of their decisions. The main drawback is that the overall speed is that of the slowest player. An improvement to this solution is asynchronous synchronization. Each player advances asynchronously until he enters a sphere of influence of another player, then he runs the lockstep protocol to manage possible interactions. We apply the same principle in our example game: no user reveals his move before receiving the other players' commitments from the server.

(2) Secret possessions: players may have secret data, that must have been acquired and managed according to the rules. Cheat detection can be done using promises. Players commit to their secrets, and announce them as promises (committed id capabilities for committed cells), then open them. A distributed *logger service* records promises; a trusted centralized *observer service* receives and dynamically checks data; a trusted centralized *promise service* receives and checks promises and data. In our example the promise service is implemented by the Resolution protocol.

Conservative protocols that could turn optimistic As opposed to the optimistic protocols cited above, conservative protocol implementations are more frequent. However they could gain efficiency if some of their runtime checks could be deferred.

Another example of distributed pessimistic application is the Jif/split compiler [Zheng et al., 2003]. Given some Java-like sequential code in which each piece of data is annotated with its security level (for instance, high or low) together with the trust level specification for each host available, Jif/split partitions securely the provided code over the given set of heterogeneously trusted hosts. When there is no host with enough integrity to store a high integrity variable, the code is replicated. For example, a variable which both Alice and Bob trust cannot be handled neither on the host H_1 trusted only by Alice, nor on the host H_2 trusted only by Bob. However, it is possible to replicate the code over both the hosts H_1 and H_2 (the secrecy components of the security label should be taken into account as well: hosts which do not ensure enough secrecy only keep the hash replicas of the variable). The code is then run in parallel on both of the hosts and the results are compared. As Alice trusts H_1 and Bob trusts H_2, if both hosts agree on the result then both Alice and Bob can trust the result and the integrity of the replicated variable is respected. Interestingly, the protocol automatically generated by the splitter to share a variable between replicas is similar to value commitment: secret values are hashed when sent to untrusted parties.

With the optimistic approach, this high integrity variable can be located on either of the half-trusted hosts and thus need not be replicated since all the transactions are securely logged. For instance, the computation on the shared variable may take place on the host H_1 which is only trusted by Alice. The necessary condition is that the computation is recorded in a secure log that Bob can access and check. The code is only run once and the verifications are done at Bob's request, so that the application gains efficiency.

3.4.3 Implementations of secure audit logs

The security of audit logs implementation is crucial. Some of the common security requirements for log systems are listed below; Other desirable high-level properties, as well as a formal framework for validating existing log systems, applied to examples, are described by Etalle et al. [2007].

- *Correctness* should ensure that if we log some data an entry corresponding to the data will be added correctly to the log.
- *Forward integrity* [Bellare and Yee, 1997] of an audit log guarantees that once logged, an entry will not be corrupted even if the machine get compromised, or, any past evidence cannot be erased by an intruder. This notion subsumes tamper resistance and verifiability [Waters et al., 2004].
  - *Tamper resistance* implies the inability of an attacker to create valid log entries or to alter the existing ones. In practice, one cannot help deleting log entries but can detect it.
  - *Verifiability* allows a public or trusted verifier to check that all created log entries are present and valid.
- *Forward secrecy* is often required when sensible data is logged. It guarantees that encryption used to protect the content remains valid after an attacker's intrusion in the system.

These requirements could be met easily by using a physically secure and tamper-resistant machine trusted by all the participants (called "trusted third party", or TTP, in what follows). Forward integrity is often achieved by remote logging to a host hardly reachable by the intruder, or by log replication thus multiplying sources of evidence which the intruder aims to erase. Another technique is to record the log entries on some kind of write-once-read-multiple physical device, or sending them to a secure printer. It is however essential to minimize the dependence on trusted parties in a log system. Instead, cryptography allows to guarantee many of these properties; we briefly present some secure log implementations. We use similar but simpler techniques in our formal implementation of committed cells.

Schneier and Kelsey [1999] propose a system where the attacker cannot seamlessly alter or delete log entries made before the logging machine is compromised. They assume the availability of a trusted third party. Before starting a new log the machine establishes a shared secret key A_0 with the TTP. The key A_{i+1} is obtained by incremental cryptographic hashing of A_i; A_i is deleted immediately after. If the machine is compromised while using the key A_k, the intruder is still unable to find the keys A_j for all $j < k$ and so to tamper with the corresponding entries. Then each i-th piece of data D_i to be logged is encrypted with a key K_i derived from A_i for the data type of D_i, thus a user in possession of K_i cannot alter the entry but just read it. The resulting bitstring is recorded as the i-th log entry. This key management guarantees tamper resistance of the scheme.

To obtain verifiability and bind all log entries, a *hash chain* Y_i for the entry i is computed as the cryptographic digest of the encrypted data concatenated with Y_{i-1}. Each record is also protected by the MAC of the hash chain under the key A_i. Even an untrusted user can thus check the validity of the hash chain. To access the log, a user addresses a request including an index range and the corresponding access permissions to the TTP. After an authorization check, the TTP issues the decryption keys corresponding to the requested log entries.

Xu et al. [2005] adopt a client-server architecture where the server has access to a trusted computer base (TCB). Initially, the TCB contains two pairs of public/private keys for encryption/decryption and signing/verifying, and a random number RN. The server adds the clients' log requests to the log as follows. A log file starts with RN, the name and the signature of the previous log file. Each record includes a sequence number, the identifier of the author, the client's data as cleartext, and the secure hash of the record computed using RN. At any time TCB contains the current sequence number and the plain accumulated hash of the log file

(PAH) to guarantee the verifiability. The PAH and its signature using the server's private key and server's public key are written at the end of the file. If restarted, the server can complete the last opened log file by checking the sequence numbers and PAH of all records. The validation of the file consists in simply checking the final signatures. This scheme shifts the basis of trust from a trusted third party to the trusted computer base of the server.

In practice it is useful to be able to find the log entries satisfying some criteria. In addition to secrecy and forward integrity properties, Waters et al. [2004] design a log system that provides searchability. The authors use identity-based encryption with extracted keywords to allow search actions on the encrypted log data.

3.5 Conclusions and future work

An optimistic approach to security offers several advantages over classical, conservative security schemes for protocols. Applications gain in efficiency by performing the necessary checks only at the end (possibly saving a large number of intermediate communications) and the enforced policies can be made more flexible (possibly improving the confidentiality of the computation). However the formal properties of log-based implementations are more delicate to formalize, and weaker than those of the corresponding pessimistic protocols. Therefore they should be implemented with particular care, the most important question being whether the evidence they log is sufficient to satisfy the claimed properties.

The results of our study of value commitment and offline e-cash schemes involve stating and proving a novel property relating the labelled traces of an abstract semantics to those of their realistic implementation: *optimistic security* – at every step, an optimistic implementation must either simulate the behaviour of the abstract model, or an illegal action happens and then it will be detected and punished.

Value commitment Our implementation of optimistic commitment has the advantage of simplicity, using only ordinary communications and standard symbolic cryptography: we use hashes to ensure secrecy of the committed value, and public-key signatures to bind a cell to the identity of a principal. We only consider authenticity for now, but we believe it would also be possible to guarantee some properties of formal secrecy.

Although committable cells provide a reasonably useful (and formally challenging) block for building protocols, we focused on one particular usage of secure logs, rather than proposing a comprehensive language design for optimistic protocols. Our formal approach proved to be extensible to other, more involved datatypes—as long as we can represent their live cycles using a preorder on exported capabilities, as detailed in Section 3.2.4.

E-cash We experimented with a similar but quite different scheme, offline e-cash. This time we used an existing cryptographic implementation [Camenisch et al., 2005].

We believe that this work brings two benefits. First, our formal semantics seems general enough to faithfully model the properties of offline e-cash protocols; we do not retain any implementation-dependent feature in our design. A possible future work in this direction is applying our approach to such e-cash protocols as Camenisch et al. [2007b]. Then, our approach to relating the low-level implementation and the abstract semantics seems useful to explore properties of other complex protocols, such as e-voting.

Our case studies thus provide a first step towards a better understanding of the usage of audit logs and similar mechanisms.

Chapter 4

A general definition of auditability

4.1 A language-based approach to auditing

Consider the mail authentication protocol from Section 2.2.1, where $\mathcal{A}$ wants to send an authenticated mail to $\mathcal{B}$:

$$\mathcal{A} \longrightarrow \mathcal{B} : \mathit{authentic}(\mathit{text}, \mathcal{A})$$

To prove her identity, $\mathcal{A}$ can sign the message using her secret signing key and append the signature to the message.

$$\mathcal{A} \ \rightarrow \ \mathcal{B} : \ \mathit{text} \,|\, \{\mathit{text}\}_{sk_{\mathcal{A}}}$$

The server $\mathcal{B}$ can verify the signature using $\mathcal{A}$'s public key and, if the test succeeds, $\mathcal{B}$ can be sure of the authenticity of the message. But, in case of dispute between $\mathcal{A}$ and $\mathcal{B}$, does $\mathcal{B}$ possess enough evidence to prove authenticity to a third party?

We say that a protocol is *auditable* with respect to a property if it logs enough evidence to convince an *impartial* third party, called a *judge*, of that property.

In our example, $\mathcal{A}$'s text and signature, if securely stored by $\mathcal{B}$, constitute sufficient evidence for auditing. Later, a judge can take a decision upon verifying the signature and, inasmuch as all principals agree on the public key infrastructure for signing, they also agree that this judge is impartial. Note that the signature alone may not constitute sufficient evidence: a careless server that discards or alters the received text would not be able to convince the judge.

Suppose now that, instead of signing the text, $\mathcal{A}$ signs a fresh key k, encrypts it under $\mathcal{B}$'s public key, and encrypts the text under k using non-malleable encryption

$$\mathcal{A} \ \rightarrow \ \mathcal{B} : \ \{k|\{k\}_{sk_{\mathcal{A}}}\}_{pk_{\mathcal{B}}} \,|\, \{\mathit{text}\}_k$$

In this case, $\mathcal{B}$ can decrypt and authenticate the key k, then decrypt the message, and infer the authenticity of *text*. However, an impartial judge cannot attribute the message to $\mathcal{A}$, since both $\mathcal{B}$ and $\mathcal{A}$ are able to encrypt data using the key k; the authenticity of *text* for $\mathcal{A}$ is not auditable. (For mail, this feature is often called *deniability* [Roe, 1997].)

The concept of auditability is entangled with the figure of the judge. A judge is an entity that evaluates if some evidence enforces a given property, in an impartial and transparent manner. Thus, its decision procedure must be relatively simple, and it must be known and accepted *a priori* by all principals concerned by the auditing. Conversely, the principals do not trust one another to comply with the protocol definition.

Similarly, fair non-repudiation protocols rely on trusted third parties (TTPs): for each message, evidence of its origin and receipt of its dispatch is collected by the participants and this evidence can be passed to the TTP to resolve disputes [Kremer et al., 2002]. Judges are similar to offline TTPs: they are invoked *a posteriori*, only when necessary. However, judges never issue their own signatures (unlike for instance transparent TTPs), nor actively participate in the protocol (for instance by sending messages to the participants).

In practice, most applications selectively store information, in audit logs, about why authorizations were granted or why services were provided, with the hope that this data can be later used for regular maintenance, such as debugging of security policies, as well as conflict resolution. However, deciding which evidence should be logged to enable reliable and efficient auditing is left to the programmer's intuition. As shown above, it is not the case that all properties that can be verified by a principal at run-time can be audited by an external judge. Even considering only properties that can be audited, it is unclear if some given evidence enforces them. Besides, extensive logging may conflict with other security goals, such as confidentiality and privacy.

This chapter proposes a formal definition of *auditable* properties (Sections 4.2 and 4.3), illustrates our definition on several sample protocols (Section 4.4), and discusses related work in Section 4.5. In the following chapters we will build on this definition, and we will study how types can be used to check if a property is auditable in a program.

4.2 Modelling security protocols in F7

We aim to verify concrete protocol implementations, rather than their abstract models, so we represent protocols as programs written in F#, a dialect of ML, and we specify their properties using logical formulas. In Section 2.4 we reviewed RCF, the formal core of the language F# and of the associated F7 type system.

A protocol can be written in F# as a collection of functions that represent compliant code for the different roles, possibly sharing some variables (such as cryptographic keys). This collection of functions and variables can be structured into modules; the module interfaces are then made available to the environment, which can run, and interact with, the roles. The environment models an active attacker; it is *a priori* untrusted and should not access some of the shared variables (such as private keys). In F# the visibility of variables is specified in typed interfaces; variables which are not exported are marked as **private**.

A *protocol*, denoted $\langle \mathcal{L}, I_{\mathcal{L}} \rangle$, is a module that defines the corresponding global variables and the roles, and its typed interface. We denote $public(I_{\mathcal{L}})$ the set of values that are exported to the adversary, and $private(I_{\mathcal{L}})$ the other values. An *opponent*, denoted O, for a protocol is an expression that does not contain any **assert** (and **audit**, defined later) and whose free variables cannot be bound to variables not declared in the public interface of this protocol. A *program* is a closed expression of the form $\mathcal{L}[O]$ where $\mathcal{L}$ defines the global variables and roles and O is an opponent (as such, it holds that $private(I_{\mathcal{L}}) \cap fv(O) = \varnothing$).

To illustrate our setup, in Figure 4.1 we program the authenticated mail of Section 2.2.1 relying on RSA public-key signatures. We call the corresponding module $\mathcal{L}_{mail}$ and its interface I_{mail}. We omit the standard library modules $\mathcal{L}_{Data}$, $\mathcal{L}_{Crypto}$ and $\mathcal{L}_{Net}$ it depends on. The interface I_{mail} exports both roles and $\mathcal{A}$'s public key; the secret key is not in the public interface of the protocol to prevent the environment from signing messages.

$$public(I_{mail}) = \{pka,\ princA, princB\}$$

The code first defines the secret key *ska* and the verification key *pka* for principal $\mathcal{A}$. The principal $\mathcal{A}$, implemented by *princA*, establishes a TCP connection c to some predefined port p, creates a signed message and sends it over c. The principal $\mathcal{B}$, implemented by *princB*, receives the message and its signature, and verifies if the signature is valid for the message issued by $\mathcal{A}$. The predicate $ASent(x)$ encodes at the logical level that the principal $\mathcal{A}$ sent the message x. Since the principal $\mathcal{A}$ is compliant, all the other participants trust her to add $ASent($`"Hey"`$)$ to the set of valid formulas using the **assume** primitive. If the signature verification succeeds, then $\mathcal{B}$ can expect this property to hold unless the secret key of $\mathcal{A}$ has been compromised (so it is now public: $Pub(ska)$). This is specified by asserting the predicate $AMightSend(x)$ which abbreviates the disjunction of these.

Figure 4.1: Code for the example 2.2.1

```
let ska = rsa_keygen()
let pka = rsa_pub ska

let princA () =
    let c = Net.connect p in
    let text = utf8 (str "Hey") in
    assume (ASent(text));
    let s = rsa_sign ska text in
    let w = concat text s in
    Net.send c w

let princB () =
    let c = Net.listen p in
    let w = Net.recv c in
    let (m,s) = iconcat w in
    if rsa_verify pka m s then
      (assert(AMightSend(m)); m)
    else failwith "Bad signature"

assume ∀m. AMightSend(m) ⇔ (ASent(m) ∨ Pub(ska))
```

Pinpointed expressions To formalise auditability we need to track precisely the substitutions that are applied to some sub-expressions of a program. Technically, we extend the syntax of expressions with *pinpointed expressions*, denoted $\underline{A}\sigma$, where σ is a finite substitution of values for variables. The definition of substitution used for evaluation is then modified to extend σ rather than propagate through A:

$$(\underline{A}\sigma)\{M/x\} = \underline{A}(\sigma;\{M/x\}) \ .$$

Once a pinpointed expression gets in head position inside an evaluation context, the deferred substitution σ is applied to A, resuming the computation via the rule $\underline{A}\sigma \to A\sigma$. Just before this reduction, σ contains exactly the substitutions applied by the context to the sub-expression A. It is easy to see that the expression A and the expression obtained by replacing a sub-expression A' of A with $\underline{A'}$ (a pinpointed expression with an empty substitution) reduce to the same value.

We recast the definition of RCF safety using pinpointed assertions:

Definition 4.2.1. The formula C is *safe* in the program $A[\textbf{assert}\ C]$ when, for all reductions

$$A[\underline{\textbf{assert}\ C}] \to^* E[\underline{\textbf{assert}\ C\sigma}]$$

where E is an evaluation context with assumed formulas F, we have $F \vdash C\sigma$. A program is *safe* when all its assertions are safe. A protocol $\mathcal{L}$ is *robustly safe* if, for all opponents O, the program $\mathcal{L}[O]$ is safe.

Note that when **assert** is evaluated the substitution σ records the actual values for the free variables of the formula C.

For example, using the protocol $\mathcal{L}_{mail}$, the program $\mathcal{L}_{mail}[princA\ ()\,|\,princB\ ()]$ is safe. The only occurrence of **assert** is in the code of *princB*, and it is evaluated after reception of the message over the connexion established on port p. Only *princA* sends a message through this port, with content `"Hey"`, and only after assuming $ASent(\texttt{"Hey"})\}$. These reductions lead to a configuration

$$E\left[\textbf{assert}\ (AMightSend(text))\{^{\texttt{"Hey"}}/_{text}\}\right]$$

with multiset of assumed formulas $F = \{ASent(\texttt{"Hey"}),\ \forall m.\ AMightSend(m) \Leftrightarrow (ASent(m) \vee Pub(ska))\ \}$, so we have $F \vdash AMightSend(\texttt{"Hey"})$. Since the private key of $\mathcal{A}$ is neither exported statically nor leaked dynamically by the protocol, it can proved as a theorem that the key remains secret in all protocol runs. So $\mathcal{B}$ can assert a stronger property: $ASent(text)$.

More interestingly, $\mathcal{L}_{mail}$ is also robustly safe. Since robust safety quantifies over all environments that interact with the protocol, we might imagine a malicious opponent that after launching *princB* establishes a connection on port p and sends the message (`"Hey"|"bleah"`). However, the signature verification performed by $\mathcal{B}$ guarantees that the received message has been sent by $\mathcal{A}$, and in turn that the formula $ASent(\texttt{"Hey"})$ has been previously assumed.

4.3 A definition of auditability

Informally, a program is *auditable* if, at any audit point, an impartial judge is satisfied with the evidence produced by the program.

We extend RCF with the primitive **audit** C M:

A ::=	expressions
...	
audit C M	audit formula C using evidence M

This allows the programmer to specify the program points that require auditing for property C, using the value M as evidence. In practice, although we do not enforce it, the evidence M should be safely logged by the program. Similarly to **assert**, this primitive plays a role only in the specification of properties: **audit** C M always reduces to unit.

To simplify the presentation, we focus on programs with a single audited property, a single judge, and a single audit request point. Let C be this property, and suppose that $fv(C) = \tilde{x}$. Our definitions generalize easily to several distinct properties and audit requests, possibly sharing the same judge.

We represent the judge as a function, named *judge*, taking as arguments the actual values of the free variables of C and the evidence, and evaluating a boolean expression J that computes the judge's decision. The judge function in a protocol should be defined by a public binding of the form **let** *judge* $\tilde{x}$ $e = J$. For sanity, we require that J does not assume any property or access any private binding of the protocol.

Auditability for the authenticated mail We already suggested that in the authenticated mail example the property $AMightSend(m)$ is not only safe but also auditable. For a given text sent by the client (e.g. `"Hey"`), the associated signature constitutes the evidence to enforce the property $AMightSend(m)\{\texttt{"Hey"}/\ m\}$. We can then replace the **assert** $(AMightSend(m))$ executed by *princB* with the audit request **audit** $(AMightSend(m))$ *sign*. In this example the PKI is trusted by all participants: a judge that, given a text and a signature, returns **true** if and only if the signature is valid can be deemed impartial (or *correct*). Observe that the signature always suffices to convince the judge: we say that it constitutes *complete* evidence.

The key property that distinguishes auditing from asserting properties, is that the judge can be called in any context where the public key of the client is known: for instance, a third party can invoke the judge to confirm the outcome of the transaction.

We can update the code of the authenticated mail protocol and add the definition of the judge.

```
let judge text e =
  verify_sig pka text e
```

```
let princB () =
  let c = Net.listen p in
  let w = Net.recv c in
  let (m,s) = iconcat w in
  if rsa_verify pka m s then
    (audit (AMightSend(m)); m)
  else failwith "Bad signature"
```

The judge function just validates the signature passed in as evidence. As discussed above, it is correct for the property $AMightSend(m)$. The **audit** $(AMightSend(m))$ *sign* statement executed by *princB succeeds* if the evidence *sign* suffices to convince the judge, as is the case here. Thus, the property *AMightSend* is *auditable* in this example. The principal *princB* then publishes the evidence on the channel *d*.

Auditability, formally Given a program $\mathcal{L}[O]$, we rewrite it as a two-hole context applied to the body J of the judge (**let** *judge* $\tilde{x}$ $e = J$) and to the evidence M provided in the audit

statement. With a slight abuse of notation we denote it as $A[J, M]$. Our definition says that a (well-formed) program is *auditable for a property* C if it defines an impartial judge for C (correctness), and if the evidence provided in the audit call suffices to convince the correct judge of the validity of the property (completeness).

Definition 4.3.1. Let $\langle \mathcal{L}, I_{\mathcal{L}} \rangle$ be a protocol with a (public) declaration **let** *judge* $\tilde{x}$ $e = J$ and a statement **audit** C M in its scope. Let O be an opponent such that $private(I_{\mathcal{L}}) \cup fv(O) = \varnothing$. Let A be a two hole context such that $A[J, M] = \mathcal{L}[O]$. The program $\mathcal{L}[O]$ is *auditable* when

(Well-formedness) (a) the declared variables of $\mathcal{L}$ are not rebound; (b) J does not contain **assume**s. (c) $fv(J) \cap private(I_{\mathcal{L}}) = \varnothing$;

(Correctness) if $A[\underline{J}, M] \rightarrow^* E[\underline{J\sigma}]$ for some evaluation context E with assumed formulas F, and $J\sigma \rightarrow^*$ **true**, then we have $F \vdash C\sigma$; and

(Completeness) if $A[J, \underline{M}] \rightarrow^* E[\underline{M\sigma}]$ for some evaluation context E, then we have $J\sigma\{M/e\} \rightarrow^*$ **true**.

The protocol $\langle \mathcal{L}, I_{\mathcal{L}} \rangle$ is *auditable* when the program $\mathcal{L}[O]$ is auditable for all opponents O.

Let us illustrate the definition above for the authenticated mail protocol, with some opponent code that receives the audit evidence on channel d then invokes the judge:

$$\mathcal{L}_{mail}[princA\ ()\ \Rsh\ princB\ ()\ \Rsh\ (\textbf{let}\ text,e = recv\ d\ \textbf{in if}\ not\ (judge\ text\ e)\ \textbf{then}\ \texttt{"bad"})]$$

With this particular opponent, the judge is called after the server successfully completes, and thus after the client's **assume**, so the judge is *correct* when it returns **true**. The evidence is also *complete*: at the audit point, if we pass the actual evidence to the judge we get

$$(verify\_sig\ pka\ text\ sign)\sigma$$

for some substitution σ that substitutes `"Hey"` for *text*, the result of *rsasha1 ska text* for *sign*, a cryptographic function for *verify_sig*, and a matching keypair for *ska* and *pka*. This expression reduces to **true** by the definition (and the F# implementation) of the verification of asymmetric signatures.

In some cases the conditions required for correctness can be trivially satisfied. A judge that always returns false is correct; however in this case no evidence can satisfy the judge, and thus the protocol cannot be complete. Also, if the judge is not called, then correctness is vacuously satisfied. Correctness and completeness are complementary properties: giving evidence to an unreliable judge makes no sense, nor does conducting a trial with insufficient evidence. Note that a judge is correct if and only if it is safe to assert the audited property whenever the judge returns true.

Extending the definition for multiple properties/audit points There is one judge function per auditable property. If there are several audit requests for the same property, Definition 4.3.1 applies for each of the audit requests in turn, while the other requests **audit** C L are replaced with **assert** C. If there are several auditables properties, Definition 4.3.1 applies independently for each of them. So there must be a judge function defined for each property, and one or several audit requests which have to be handled as described above. Audit requests for other properties can be safely replaced with **assert**s when applying the definition for one particular property.

Syntactic convention The current implementation of F7 does not support pattern-matching on formulas. To circumvent this syntactic restriction in the code snippets we write ad-hoc functions to embed the audit requests; so we write *auditPredicate* $\tilde{x}$ *evidence* rather than **audit** *Predicate*$(\tilde{x})$ *evidence*. Accordingly we name the corresponding judge function *judgePredicate*.

For our formalization (Section 5 particularly) we continue using the abstract **audit** primitive, and consider a single auditable property and a single audit request.

Opponents and partial compromise The environment models a potentially hostile attacker, which can access all public values and roles of the protocol, the cryptographic library (to model its knowledge of the cryptographic algorithms), and to the network (to model the control of communications by active adversaries). In addition, an attacker may corrupt a subset of the principals to gain access to their private resources (like signing keys). Interestingly, in this case the remaining compliant principals may remain auditable: a signature by a principal, compliant or not, constitutes audit evidence.

Compromised participants can be represented in our setting by extending the protocol with definitions that export their private resources. Suppose that, in the authenticated mail example, $\mathcal{A}$ is compromised. Its secret key becomes public and is typed as $ska{:}key\{Pub(ska)\}$, and the code below is added to the end of the protocol:

let *leaked_key* = **assume** $(\forall x.\ ASent(x))$; *ska*

The attacker can now choose any message and sign it with $\mathcal{A}$'s signature:

let *bad_text* = *utf8* (*str* "Bleah") **in**
send c (*bad_text*, *rsa_sign leaked_key bad_text*)) ↱ *princB* ()

The compromise of $\mathcal{A}$ must be reflected in the logical world. The meaning of the formula $ASent(x)$ was that "principal $\mathcal{A}$ sent message x", and it was possible to certify this action by verifying the relevant signature. However, now arbitrary messages sent by the attacker can be signed with $\mathcal{A}$'s key. The **assume** $(\forall x.\ ASent\ (x))$ evaluated just before exporting the private key of $\mathcal{A}$ captures this fact. In general, before exporting the private resources of a compromised participant, it is necessary to either "saturate" all the properties related to the compromised participant, as done here (in a modal logic, this would be equivalent to assuming the formula $\mathcal{A}$ **says false** [Fournet et al., 2007]) or assume some compromise at the protocol level. Observe that, in a protocol run where $\mathcal{A}$ was compromised and the environment issued the attack above, if an audit for the property $ASent(bad\_text)$ is requested, then the server can still provide enough evidence to the judge: the protocol is still auditable.

An attacker might also invoke directly a judge and provide some bogus evidence to accuse a compliant principal. However, Definition 4.3.1 states that a judge is correct only if it always takes the right decision, independently of the origin of the evidence. So, this attack is deemed to fail.

4.4 Auditability, illustrated

Definition 4.3.1 encompasses the properties of our target examples, but also rules out non-auditable properties.

4.4.1 Naive (non-auditable) mail

For non auditable protocols there is no such judge function that is correct and complete. We illustrate this using the naive mail protocol, which is neither auditable nor authentic.

$$\mathcal{A} \to \mathcal{B} : text$$

The protocol may be written as $\mathcal{L}$:

let *roleA txt* = **assume** *Auth*(*txt*); *send c txt*
let *roleB* () = **let** *txt* = *rcv c* **in audit** *Auth*(*txt*) *txt*

Formula *Auth*(*txt*) models the authenticity of the message. The audit request at $\mathcal{B}$'s side collects all the received evidence, here *txt*.

Proposition 4.4.1. *The protocol $\mathcal{L}_{naive}$ is not auditable for property Auth(txt).*

PROOF: Suppose that there is a function **let** *judge x e* = *J* such that *Auth*(*txt*) is auditable for the protocol $\mathcal{L}'$ = (**let** *judge x e* = *J* $\mathcal{L}$).

(1) First, we consider the following opponent run:

$\mathcal{L}'$ (*roleA* `"Yes"`↱ *roleB* ()) →* (**assume** *Auth*(`"Yes"`) ↱ **audit** *Auth*(`"Yes"`) `"Yes"`).

By completeness, we have that *J*{`"Yes"`/*x*,`"Yes/e`} →* **true**.

(2) Now, consider a different opponent run:

$\mathcal{L}'$ (*roleA* `"No"`↱ *roleB* () ↱ (**let** _ = *rcv c* **in let** *x*,*e* = `"Yes"`,`"Yes"` **in** *send c x*; *J*))

→* (**assume** *Auth*(`"No"`)↱ **audit** *Auth*(`"No"`) `"No"` ↱ *J*{`"Yes"`/*x*,`"Yes"`/*e*})

The judge is called in the same configuration, as in (1), except for the assumed formulas which are irrelevant for the reduction semantic so, so from (1) we know that

J{`"Yes"`/*x*,`"Yes"`/*e*} →* **true**.

By the assumed correctness of the judge *Auth*(`"Yes"`) must hold, however it does not. We obtain a contradiction, so *Auth* is not auditable for this protocol. □

4.4.2 Rock-Paper-Scissors

We consider a simple implementation of the Rock-Paper-Scissors protocol (described in 2.2.2) where the bid authentication is achieved via RSA signatures (term *proof* is an RSA digital signature). A trusted server organizes a game between $\mathcal{A}$ and $\mathcal{B}$ who run the role *princ* parametered with their identity, the port number used to communicate with the server and their bid.

```
type bid = Rock | Scissors | Paper

assume (Beats(Rock,Scissors))
assume (Beats(Scissors,Paper))
assume (Beats(Paper,Rock))


assume ∀p,x. MightSend(p,x) ⇔
            (Sent(p,x) ∨ Bad(p))
assume (∀p1,p2,x1,x2. (MightSend(p1,x1) ∧
    MightSend(p2,x2) ∧ Beats(x1,x2)) ⇒ Wins(p1
    ))

assume ∀p,b,x. SendFrom(usage,p,b) ⇔
              (Sent(p,x) ∧ BytesOfBid(b,x))

let beats x y =
  match x with
  | Rock → y=Scissors
  | Scissors → y=Paper
  | Paper → y=Rock
```

```
let judgeWins w (ma,sa,mb,sb) =
  let ba = bid2bytes ma in
  let bb = bid2bytes mb in
  if rsa_verify_prin usage alice pka ba sa
  then if rsa_verify_prin usage bob pkb bb sb
  then begin
    assert(MightSend(alice,ma));
    assert(MightSend(bob,mb));
    if beats ma mb then w=alice
    else if beats mb ma then w=bob else false
  end


let princ p port bid =
    let c = Net.connect port in
    let text = bid2bytes bid in
    assume (Sent(p,bid));
    let sk = getPrivateKey usage p in
    let s = rsa_sign usage p sk text in
    let w = concat text s in
    Net.send c w
```

Sum type *bid* ranges over possible bids: *Rock*, *Paper* or *Scissors*; injective functions *bid2bytes* and *bytes2bid* allow un/marshalling.

Predicate *Beats*(*x*,*y*) records that bid *x* beats the bid *y*. The usual game rules are specified as assumptions trusted by all participants. Although all formulas in our examples so far are just facts (representing protocol events), in general formulas also include policy rules. Predicate *Sent* (*p*,*x*) records that a player *p* made the bid *x*; predicate *MightSend*(*p*,*x*) abbreviates the weakened version of this, considering the compromise of the principal *p*. The main rule for recognizing a victory of a principal, denoted with predicate *Wins*(*p*), states that this principal must have

made a bid, and another principal must have made a weaker bid. Note that formulas like *Sent* are just facts representing runtime protocol events, while formulas like *Beats* are policy rules.

During the protocol both players send their signed moves to the server who determines the winner. We show that the victory *Wins* claimed by the server can be audited. Intuitively, a judge needs the moves, together with their signatures, for both players, to decide if the proposed winner actually won the game.

The judge will be called in a context that assumes the global rules of the game. To be correct, its decision must respect these rules. The judge returns **true** only after verifying the move signatures (which in turn guarantees that the properties *Send*(*alice*,*Paper*) and *Send*(*bob*,*Rock*) holds), and *w* is the player who played the winning move (computed by the function *beats*). So the judge is correct for the property *Wins*. Alternatively, the only audit request occurs after the verification of the signatures passed as evidence and calling the function *beats*. So we have both correctness of the judge and completeness of the provided evidence.

4.4.3 Value commitment

In Section 3.2 we showed how to audit write-after-commit cheating in the implementation of committable cells. Committed id and rd capabilities are represented with records including the relevant cryptographic fields.

type *idc* = {*ownidc*:*prin*; *hidc*:*bytes*; *sigidc*:*bytes*}
type *rd* = {*ownrd*:*prin*; *id*:*id*; *value*:*value*; *sigrd*:*bytes*}

The auditable property of this schema is "Principal *p* is guilty of multiple-commit of a cell", expressed as predicate *MultCommit*(*p*). Predicate *HAssign*(*p*,*hid*,*hv*) records the intention of principal *p* to commit to some value *v* in cell *c*, which corresponds to the resulting committed id capability containing an hashed value *hv* for the hashed identifier *hid*. To reliably verify this property, it is sufficient to be given two committed id capabilities belonging to this principal, for the same cell and check that they are related to the same cell, are well-formed and have different contents. The main rule for blaming and the code of the judge are shown below:

assume $\forall p,id,v,v'.\ (HAssign(p,id,v) \wedge HAssign(p,id,v') \wedge v \neq v') \Rightarrow DoubleSpent(p)$

```
let check_idc i =
  let vkp = getPublicKey usage i.ownidc in
  rsa_verify_prin usage i.ownidc vkp i.hidc i.sigidc

let judgeMultCommit p (cap1,cap2) =
  let _ = getPublicKey usage cap1.ownidc in
  let _ = getPublicKey usage cap2.ownidc in
  if cap1.ownidc = p then
  if cap1.ownidc = cap2.ownidc then
  if check_idc cap1 then
  if check_idc cap2 then
  let (h1,h1') = iconcat cap1.hidc in
  let (h2,h2') = iconcat cap2.hidc in
  if h1 = h2 then
  if h1' ≠ h2' then true
  else false
```

We can then show that the distributed resolution process blaming a principal is auditable for this judge function.

Independently, we can show that the authenticity of a cell content in this implementation of value commitment is auditable. The judge function simply checks the signature of the owner, given the identity, the content, and a committed capability (read or committed id) of the cell.

4.5 Discussion and related work on auditability

"Positive" and "negative" properties As the value commitment example illustrates, the same program may have several audit goals of different kinds. Standard – pessimistic – protocols use logs as an additional means to validate their security and make it provable to third parties. The auditable properties often express "positive" statements: a participant obeyed the game rules, the decision taken by a participant is fair, etc.

Optimistic protocols usually rely on logs for cheat detection, so their auditable properties have a "negative" flavour and may express the correctness of the cheat detection procedure. In turn, correctness properties of the protocol itself are often disjunctions of the desired property when all participants behave honestly, and of the blame in case a principal behaves dishonestly.

Non cryptographic evidence Even if typical evidence includes some collection of signed data, the judge does not necessarily rely on cryptography. To audit the arithmetic property "$2^n - 1$ is not prime", with two integers as evidence, a correct judge simply checks that these integers are greater than 1 and their product is equal to $2^n - 1$. Similarly, if an access control database is trusted by the judge and by all principals, then the compliance of granted or denied accesses can be verified against the corresponding database entries, and no evidence must be provided.

Auditability versus other security properties As the introductory mail example illustrates, authenticity does not entail auditability in general, though auditability often subsumes authenticity (intuitively, **audit** $C\ L$ can always be safely replaced with **assert** C).

Some properties, like deniable authentication in the second example of Section 4.1, cannot be audited. (Note that a weaker property which is the disjunction saying that either $\mathcal{A}$ or $\mathcal{B}$ sent the message can be proved). In general, all deniable properties are not auditable, and all auditable properties are undeniable (luckily properties enforced by most of the protocols are neither deniable nor undeniable).

Privacy and secrecy constraints often conflict with auditing; it is well-known that "if logs mention private information they are forbidden and if they do not – they are useless" [Etalle et al., 2007]. For instance, if x is secret, then a property C where x appears as cleartext (in the property or in the evidence) cannot be audited. Protocols using zero-knowledge proofs in particular constitute an interesting source of auditable properties. Peha [1999] proposes a system design for electronic commerce where privacy of customers is taken into account to some extent; information on their transactions is only available to the system auditors, and not to the other parties. For some cases, this concession is though inacceptable.

Accountability is a related, and in some cases synonymous security property; we discuss it in the rest of this section.

Accountability for optimistic security enforcement The use of logs for optimistic security enforcement has been advocated in earlier work [Cederquist et al., 2007, Etalle and Winsborough, 2007]. The work closest to our is by Cederquist et al. [Cederquist et al., 2007]; they develop an audit-based logical framework for user accountability, specialized for discretionary access control. They also design cryptographic support for communication evidence in a decentralized setting [Corin et al., 2006]. In their framework, all auditors (judges) are based on a sound and complete proof checker, and are correct in our sense. However, principals must rely on a tamper resistant logging device to prevent a malicious agent from forging a log entry. In comparison, we delegate the integrity and authorization checks to the code of the judge. Their framework defines whether an agent is *accountable* for a given run, and hints that if an agent logs all relevant evidence before each action (following an *honest strategy*) then all of its run will be accountable.

Etalle and Winsborough [2007] propose a logical framework relying on logs that embeds a trust management system for mapping agents to roles. *Auditability* of a principal for an action implies that the principal is able to prove the compliance of his action with respect to the current policies. *Accountability* of a user refers to his legitimacy and susceptibility to the eventual punishment. Agents are required to check whether the action they are going to perform complies to the current policies (auditability) and log the corresponding evidence. In particular, all communications, creating and sending a document must be logged. Receiving a document requires checking that the sender is *trustworthy*: auditable and accountable.

Related work on secure provenance [Hasan et al., 2007] allows to reliably determine the origin of data (for instance, the place of residence of a cow that has mad cow disease can be used to track down the source of propagation). A provenance certificate is a standalone set of records that includes cryptographically encrypted or signed data and keying material, and provides integrity and selective secrecy for the data . Both audit trails in our approach and provenance certificates can be seen as proof verifiable out-of-context.

Accountability from protocol perspective Küsters et al. [2010] propose a general definition of accountability for cryptographic protocols both in symbolic and computational models. A judge (auditor) is a an agent that states verdicts on protocol runs. A verdict is a logic comibination of propositions recording the blame of principals. An accountability property Φ is a set of verdicts when the protocol run satisfies some trace property. A judge ensures symbolic Φ-accountability for a protocol if

- (fairness) the judge's verdicts are never false, in all runs
- (completeness) in all traces satisfying an accountability property from Φ, and in all runs, the verdict of the property holds.

This definition is also centered around guilt, rather than proving generic properties. Unlike in Definition 4.3.1, fairness says that the judge must be correct for all properties (in our case it only says "yes" or "no" for a given property). For completeness, the accountability properties Φ embed the execution trace of the protocol at the point where the verdict is stated by the judge. We put the audit request points directly in the code, thus also quantifying on all possible executions of the protocol.

Jagadeesan et al. [2009] propose a framework for design and evaluation of accountability systems with authorization security goals. Compliant behavior of system users is specified using CSP based processes. Illegal traces are those which cannot be produced by compliant (honest) processes, and principals issuing them are said to be dishonest. In this authorization setting, illegally sent messages constitute crimes, and auditors merely trace their provenance. An auditor is defined as ordinary honest principal, who can receive audit requests, and ask other principals to justify their actions, in order to decide which principal to blame, accordingly. An auditor may enjoy the following properties:

- (liveness) a non-empty set of agents is always blamed;
- (correctness) (1) Upper bound: every guilty agent is blamed; (2) Overlap: at least one of the blamed agents is guilty; and (3) Lower bound: every blamed agent is guilty;
- (blamelessness) honest principals can avoid being blamed.

Their definitions is built around the notion of guilt, while our definition allows an auditor to take any kind of decision (and in particular to determine the guilt of a set of agents, thus implementing one of the flavours of correctness). Practically, if the communication channels between the auditor and the agents do not provide non-repudiation and if no trust assumptions are made on agents, auditors can at best enjoy the Overlap correctness (some of the blamed agents are guilty). Using trusted third parties (notaries) allows to recover the Lower bound correctness (all blamed agents are guilty).

Accountability from operating systems perspective Much work has been done on how to use logs to track faulty behaviours of nodes in distributed systems.

Yumerefendi and Chase [2005, 2004] advocate the importance of accountability in dependable distributed systems. Problems arising from the heterogeneity of system trust assumptions across different sites can be dealt with by integrating accountability concerns in the design. The authors suggest that the actions and state of each agent (node) are undeniable, tamper-evident and certifiable (verifiable by an auditor). An auditor, like in our setting, may require an agent to prove the correctness of its actions.

PeerReview [Haeberlen et al., 2007] implements accountability for distributed systems. Nodes composing systems are modeled as deterministic state machines (the part to be accounted for) and an application part (not relevant for accountability, may contain secret data). A system node is either correct – it follows the given protocol –or faulty. Only faults observable by correct nodes are considered. If a node sends a message that a correct node would not send (a detectably faulty node), a correct node that has a proof of it can expose this faulty node. If a node does not send a message that a correct node would send (a detectably ignorant node), a correct node that does not receive the message can suspect this faulty node until it receives the message (so eventually forever). PeerReview guarantees two properties:

- (completeness) eventually, every correct node suspects forever every detectably ignorant node, and exposes or suspects one of the nodes causally affected by each detectably faulty node.
- (accuracy) no correct node is forever suspected or exposed by a correct node

Every node keeps a tamper-evident append-only log of its inputs and outputs, as well as periodic snapshots of its state. Log entries form an hash chain; some entries are authenicated with digital signatures. Commitment protocol is used to guarantee non-repudiation of origin and receipt for the logged messages. Every node is associated to a set of witnesses, nodes who collect evidence, check and distribute the verification results. The consistency protocol ensures that each node maintains a single linear log (similar to the double-commitment detection problem in Section 3.2). One of the witnesses must be correct at least to ensure completeness. A node is regularly audited by its witnesses: he receives a request two authenticated log entries, and must provide all the log entries chronologically in between them. During audit the witness runs the protocol reference implementation initiliazed with the a recent snapshot from the received log, and compares the outputs with the log; any difference allows to expose the node. The authors choose to use the reference implementation to avoid introducing a gap relative to a formal specification, or define a judge function like we do. However the amount of logged data is considerably larger, since all the input/outputs, as well as state some state snapshots, must be logged. The practical implementation of the system is costful, in particular the consistency and evidence-transfer protocol between the witnesses and the other correct nodes.

A similar idea has been applied to protect the integrity of distributed Web applications by the Ripley tool [Vikram et al., 2009]. Untrusted client-side application parts are replicated and run in parallel at the server's side, the results of both untrusted and trusted executions are logged. Ripley automatically transmits all user inputs to the replica, and compares the logs: all discrepancy exposes an attack. Though the main intention of the tool is to dynamically guarantee the application integrity – that is, the server execution blocks until the client's side results are received and successfully checked against the trusted replica's results – the tool can also be used in an optimistic way where the logs are only checked periodically.

Maniatis [2003] developed a way to provide global secure history preservation using entanglement of distributed logs in an undeniable, tamper-evident way, based on logical clocks and cryptographically secure hash functions.

Existing meanings of Auditability The term "auditability" is not new; it has been used to refer to various properties sometimes quite different from the one we propose. Below we

attempt a non-exaustive disambiguation.

As we already mentioned, according to [Etalle and Winsborough, 2007] *auditability* of a principal for an action implies that the principal is able to prove the compliance of his action with respect to the current policies. In our setting this property is implicitly assumed for all principals.

More related to our meaning, the term of *auditability* has been used in previous work to denote the informal ability of a system to produce enough evidence for a posteriori audit analysis (as equivalent to material evidence). For instance, Peha [1999] suggests that "A system for electronic transactions should be as auditable as transactions in the physical world ... it should be possible to identify the guilty parties ... even if the parties in a transaction and operators of the system itself cooperate to falsify records".

Chaum [1982] in his pioneering work on blind signatures for anonymous e-cash refered to *auditability* as to the "ability of individuals to provide proof of payment" in case of an audit; an ability that again contradicts the anonymity privilege of payers.

A company claiming auditability of its digital notary services, Surety [sur, 1994] offers a notary service for protecting and independently verifying and proving the authenticity of electronic documents. Given a customer's document, its hash is bound and timestamped to get a Surety Integrity Seal which can be then be used to prove the authenticity of the document to anyone. Each hash is integrated into the global system's hash chain whose integrity value is regularly published in the *New York Times*. The original work, underlying the company's patents, proposed a method of time-stamping a digital document [Haber and Stornetta, 1991] with a time-stamping service (TSS). It introduced several key ideas: first, for authentication purposes, it is sufficient to send the hash of the document to TSS, rather than the whole document; second, the TSS can sign the time-stamp together with the hashed document to prove its competence to the client and to avoid storing the record; third, to prevent TSS from issuing false (past- or future-dated) time-stamps, TSS should include bits of previously time-stamped documents in its signature.

Chapter 5

Automatic verification of auditability

This chapter presents an automatic method for verifying the property of auditability defined in Section 5.1. We rely on refinement types and use the F7 typechecker for F# [Bhargavan et al., 2008a] to statically verify that a property is auditable in a program. Our approach is tested against several sample protocols, including the F# implementation of a realistic multi-party partial-information game (Section 5.2). The source code for all our protocols is available online [cod].

5.1 Static analysis of auditability

In the previous chapter we rely on **assume**, **assert**, and **audit** statements to relate the states of a program to logical formulas. Our aim now is to assign precise types to the judge function and the audit statement, so that typechecking a well-formed protocol will be enough to satisfy the hypotheses of Definition 4.3.1, and thus to verify that a property is auditable.

We first discuss the typing annotations required to guarantee the correctness of the judge, then those needed to guarantee the completeness of the evidence.

Correctness A judge is a public function that returns a boolean value. The untrusted environment should be able to invoke it, so the arguments of the judge function must be of type bytestrings refined with predicate *Pub* (this type is abbreviated as *bytespub*). In particular, the evidence values themselves are not trusted until they are verified by the judge. The correctness condition requires the judge to return **true** only when the target audited property (say C) holds; this can easily be expressed as a post-condition on the return type. This suggests the following type declaration for the judge function:

val *judge*: $\tilde{x}$: $\widetilde{bytespub} \rightarrow e$:*bytespub* $\rightarrow b$:*bool* { b=**true** $\Rightarrow C$}

Any expression that can be given this type is a correct judge function.

Completeness Definition 4.3.1 states that some evidence is complete for a successful audit request if a call to the judge in the same context and with the same evidence returns **true**. This requires that: (1) the judge terminates, and (2) if the judge terminates, it returns **true**.

Termination of the judge function must be proved manually. Termination is hard to prove in general, but pragmatically we limit ourselves to judges that are sequences of calls to deterministic functions that terminate unconditionally: either non-recursive functions, or recursive functions that can be easily shown to terminate (e.g. functions over lists). For example, we assume that all functions defined by libraries *Data* and *Crypto* terminate on all inputs. So termination is not a real issue.

We must then show that the context of every **audit** provides enough guarantees on the gathered evidence to ensure that the judge returns **true**. This amounts to writing a *success condition* for the judge; typechecking is then used to verify that the condition holds at every audit point.

Let D be a formula with its free variables ranging over $\tilde{x}$, e, and the public variables of the protocol. We say that D is a *success condition* for *judge* if *judge* can be type-checked against the refined type

val *judge* : $\tilde{x}$: $\widetilde{bytespub} \to e$:*bytespub* $\to b$:*bool* { $(D \Rightarrow b$=**true**$)$ }

Observe that the success condition does not need to be trusted, as its correctness is checked by the type-checker. Guessing a valid success condition might not be easy. Typically a judge is a sequence of verifications: its success condition is the conjunction of the success conditions for each of them. A successful call to *judge* also guarantees completeness of the evidence; in practice it is helpful to record the fact that *judge* returns **true** only if the success condition holds. If we combine these post-conditions for completeness with that required for correctness, we obtain the following type annotation:

val *judge* : $\tilde{x}$: $\widetilde{bytespub} \to e$:*bytespub* $\to b$:*bool* { $(b$=**true** $\Rightarrow (C \wedge D)) \wedge (D \Rightarrow b$=**true**$)$ }

Typechecking must then guarantee that the success condition D holds for the evidence used in the actual audit request. To enforce it in the F7 code that includes the audit primitive **audit** C L, we declare that **audit** is a function typed with precondition D:

private val audit : $\tilde{x}$: $\widetilde{bytespub} \to e$:*bytespub* { D }$\to$ *unit*

This guarantees that the success condition holds in the judge invocation context. We suppose that the audit function is simply implemented as **let audit** $\tilde{x}$ $e = ()$ in all protocols which contain an audit statement.

With these type annotations, typechecking plus unconditional termination of the judge imply auditability (as shown in Theorem 5.1).

Success conditions and cryptography We also need some additional refinements for public-key signatures, so that typechecking guarantees the success of future signature verifications once a signature has been verified. We introduce a predicate $IsDsig(vk, m, sg)$ where vk, m, and sg are of type *key*, *bytes*, and *bytes* respectively. This predicate records key-data-signature triples for which the cryptographic primitive *rsa_verify* is guaranteed to succeed.

assume $\forall vk,m,sg.\ IsDsig(vk,m,sg) \Leftrightarrow \exists sk.\ PubPrivKeyPair(vk,sk) \wedge IsSignature(sg,sk,m)$

The post-condition of *rsa_verify* is now a conjunction that captures the two uses of the function: either we do not know whether it will succeed and if it returns **true** we learn one *IsDsig* fact; or we know the relevant *IsDsig* fact and we deduce that it will return **true**.

val *rsa_verify* : *vk*:*signed verifkey* $\to$ *p*:*payload* $\to$ *sg*:*dsig* $\to b$:*bool*
{ b=**true** $\Rightarrow (C \wedge IsDsig(vk, p, sg)) \wedge (IsDsig(vk, p, sg) \Rightarrow b$=**true**$)$ }

For example, our judge for authenticated mail calls *rsa_verify* once, and it can now be re-typed with a success clause:

val *judge* : *text*:*string* $\to$ *e*:*bytes* $\to b$:*bool*
{ $(b$=**true** $\Rightarrow (Send($`"A"`$,text) \wedge IsDsig(pka,text,e)) \wedge (IsDsig(pka,text,e) \Rightarrow b$=**true**$)$ }

Formalization To formally state this result we must first package the protocol as a refined module. For instance, in our mail example, we must add the following type for **audit** to the interface of the module:

private val audit : *text*:*string* $\to$ *e*:*bytespub* $\{IsDsig(pka,text,e)\}\to$ *unit*

The following theorem formalizes the methodology introduced above.

Theorem 5.1 (Auditability by typing)**.** *Let $\langle \mathcal{L}, I_{\mathcal{L}} \rangle$ be a well-formed protocol with a judge function that always terminates and an audit statement* ***audit*** *C L in its scope (with $fv(C) = \{\tilde{x}\}$).*

If $(\varnothing, \mathcal{L}, I_{\mathcal{L}})$ is a refined module
and there exists a success condition D (with $fv(D) \subseteq \{\tilde{x}, e\} \cup public(I_{\mathcal{L}})$) such that
*$I_{\mathcal{L}}(judge) = \tilde{x}$: $\widetilde{bytespub} \to e$:$bytespub \to b$:$bool\{(b$=**true** $\Rightarrow (C \wedge D)) \wedge (D \Rightarrow b$=**true**$)\}$ and*
*$I_{\mathcal{L}}($**audit**$) = \tilde{x}$: $\widetilde{bytespub} \to e$:$bytespub$ $\{$ D $\}\to$ $unit$ and*
then the protocol $\langle \mathcal{L}, I_{\mathcal{L}} \rangle$ is auditable for C.

The proof of Theorem 5.1 depends on a lemma that states that the type of the judge and audit function do not change during the reductions, despite the presence of an opponent.

Lemma 5.2 (No rebinding)**.** *For all refined module $(\varnothing, X, I)$, for all opponent O such that $I \vdash O : unit$, for any public value v such that $v \in bv(I)$, if $X[O] \to^* E[A]$ for some evaluation context E and expression A, and $\varnothing \vdash X[O] : unit$ then for all subderivation $\Gamma \vdash v : T_v$ within the type derivation $\varnothing \vdash E[A] : unit$ we have $\Gamma \vdash v : I(v)$.*

Proof of Lemma 5.2 This lemma is a variant of the subject reduction result for *RCF* (Theorem 2.1) relying on the fact that bound variables of modules are distinct. □

Proof of Theorem 5.1 Let $\langle \mathcal{L}, I_{\mathcal{L}} \rangle$ be a well-formed protocol which contains:
- a *judge* function that always terminates: **let** *judge* $\tilde{x}$ $e = J$, and
- an **audit** function: **let audit** $\tilde{x}$ $e = ()$
- an audit request **audit** $\tilde{M}$ N.

Let C and D be two properties such that $fv(C) = \tilde{x}$ and $fv(D) \subseteq \{\tilde{x}, e\} \cup public(I_{\mathcal{L}})$

We suppose that

(1) $I_{\mathcal{L}}(judge) = \tilde{x}$: $\widetilde{bytespub} \to e$:$bytespub \to b$:$bool\{(b$=**true** $\Rightarrow (C \wedge D)) \wedge (D \Rightarrow b$=**true**$)\}$

(2) $I_{\mathcal{L}}($**audit**$) = \tilde{x}$: $\widetilde{bytespub} \to e$:$bytespub$ $\{$ D $\}\to$ $unit$

(3) $(\varnothing, \mathcal{L}, I_{\mathcal{L}})$ is a refined module.

By Theorem 2.2 the module $(\varnothing, \mathcal{L}, I_{\mathcal{L}})$ is robustly safe. Hence, $\mathcal{L}$ is safe for every opponent O such that $I_{\mathcal{L}} \vdash O : unit$ and $\varnothing \vdash \mathcal{L}[O] : unit$.

Let O be such an opponent. We write $\mathcal{L}[O]$ as $A[J, L]$.

Correctness We suppose that $\mathcal{L}[O] \to {}^*E[\underline{J}\sigma]$ for some evaluation context E and substitution σ. Since $\underline{J}$ only occurs in the body of the function *judge*, we have $\mathcal{L}[O] \to {}^*E[(judge\ \widetilde{M}\ N)\sigma']$ where $\sigma' = \sigma'[$**fun** $\tilde{x} \to$ **fun** $e \to J/judge]$.

By subject reduction (Theorem 2.1), $\varnothing \vdash E[(judge\ \tilde{M}\ N)\sigma'] : unit$, where E binds the names $\tilde{a}$ and assumes the formulas F.

By the typing derivation above, we have $\Gamma \vdash (judge\ \tilde{M}\ N)\sigma' : T$ for some type T in the typing environment $\Gamma = \tilde{a}, F$.

By Lemma 5.2, we have $\Gamma \vdash judge\,\sigma' : I_{judge}(judge) = \tilde{x}$: $\widetilde{bytespub} \to e$:$bytespub \to b$:$bool\{($ b=**true** $\Rightarrow (C \wedge D)) \wedge (D \Rightarrow b$=**true**$)\}$.

By typing rule for application, we have $\Gamma \vdash (judge\ \tilde{M}\ N)\sigma' : b$:$bool\{(b$=**true** $\Rightarrow (C \wedge D)) \wedge (D \Rightarrow b$=**true**$)\}\sigma'$.

Suppose that $J\sigma \to^*$ **true**, then $E[judge\ \tilde{M}\ N)\sigma'] \to^* E[$**true**$]$. Then, by subject reduction (Theorem 2.1), $\Gamma \vdash$**true** $:b$:$bool\{$ $(b$=**true** $\Rightarrow (C \wedge D)) \wedge (D \Rightarrow b$=**true**$)$ $\}\sigma'$.

Since **true** is a value, the typing derivation above must use the value typing rule (Typ Refine). Hence, we must have the logical derivation $F \vdash (b$=**true** $\Rightarrow (C \wedge D)) \wedge (D \Rightarrow b$=**true**$)$ $\}\sigma'[$**true**$/b]$. That is, $F \vdash C\sigma'$.

Completeness We suppose that $\mathcal{L}[O] \to^* E[\underline{N}\sigma]$ for some evaluation context E that binds the names $\tilde{a}$ and assumes the formulas F and for some substitution σ. Since the term $\underline{N}$ only occurs in the function application **audit** $\widetilde{M}$ N, we have $\mathcal{L}[O] \to {}^*E[(audit\,\widetilde{M}\,N)\sigma']$ where $\sigma' = \sigma[\textbf{fun}\ \tilde{x} \to \textbf{fun}\ e \to J/judge][\textbf{fun}\ \tilde{x} \to \textbf{fun}\ e \to ()/\textbf{audit}]$ (by assumption, the audit statement is in the scope of $judge$). By subject reduction (Theorem 2.1), $\varnothing \vdash E[(audit\,\widetilde{M}\,N)\sigma'] : unit$.

By the typing derivation above, we have $\Gamma \vdash (audit\,\widetilde{M}\,N)\sigma' : T$ for some type T in the typing environment $\Gamma = \tilde{a}, F$.

By Lemma 5.2, we have $\Gamma \vdash audit\,\sigma' : I_{audit}(audit) = \tilde{x} : \widetilde{bytespub} \to e{:}bytespub\{D\} \to b{:}unit$.

By typing rule for application, we have

$$\Gamma \vdash \tilde{M}\,\sigma' : \widetilde{bytespub} \tag{5.1.1}$$

$$\Gamma \vdash N\,\sigma' : e : bytespub\{D\}\sigma' \tag{5.1.2}$$

We consider the type of the expression $J\sigma'\{N/e\}$ in the environment Γ. Since J occurs in the body of $judge$, we have $J\sigma'\{N/e\} \equiv (judge\,\widetilde{M}\,e)\sigma'\{N/e\} \equiv (judge\,\widetilde{M}\,N)\ \sigma'\{N/e\}$.

By Lemma 5.2, we have $\Gamma \vdash judge\,\sigma'\{N/e\} : I_{judge}(judge) = \tilde{x}{:}\ \widetilde{bytespub} \to e{:}bytespub \to b{:}bool\{(b{=}\textbf{true} \Rightarrow (C \wedge D)) \wedge (D \Rightarrow b{=}\textbf{true})\}$.

We can apply the typing rule for application to the arguments $\widetilde{M}$ and N (5.1.1 and 5.1.2), and obtain $\Gamma \vdash (judge\,\tilde{M}\,N)\sigma'\{N/e\} : b{:}bool\{(b{=}\textbf{true} \Rightarrow (C \wedge D)) \wedge (D \Rightarrow b{=}\textbf{true})\}\sigma'\{N/e\}$.

By typing rule for values and 5.1.2, we have $\Gamma \vdash N\,\sigma' : (D\{N/e\})\sigma'\{N/e\} = D\sigma'\{N/e\}$.

By typing rule for refined values, $\Gamma \vdash (judge\,\tilde{M}\,N)\sigma'\{N/e\} : b{:}bool\{D \wedge (b{=}\textbf{true} \Rightarrow (C \wedge D)) \wedge (D \Rightarrow b{=}\textbf{true})\}\sigma'\{N/e\}$.

After simplifying the formula we have $\Gamma \vdash (judge\,\tilde{M}\,N)\sigma'\{N/e\} : b{:}bool\{b{=}\textbf{true}\}\sigma'\{N/e\}$.

By assumption, the judge terminates, so there exists a value V such that $E[judge\,\tilde{M}\,N\,\sigma'] \to^* V$.

By subject reduction, we have that $\Gamma \vdash Vb{:}bool\{b{=}\textbf{true}\}$.

By typing rule for values, we must have the logical derivation $F \vdash b{=}\textbf{true}\{V/b\}$. So $V =$ **true** in the evaluation context E. □

5.2 Application: a protocol for n-player games

We implement and verify security properties of the multi-party game protocol described in Section 2.2.3. Interestingly, it offers a non-trivial auditable property: at the end of the game, depending on the moves for all players, one player wins and can prove his wins to be recognised as the winner.

Informal description of the protocol We review the protocol from the perspective of a concrete ML implementation.

The protocol has two roles, the player and the server; each run involves $n + 1$ principals, n players plus one server. The same principal may be involved multiple times in the same run, as several players plus possibly the server.

The protocol has three rounds, each with a message from every player to the server, followed by a message from the server multicast to every player:

$Player_i \longrightarrow Server : Player_i$	Hello
$Server \longrightarrow Player_i : id, \widetilde{Player}, \{id, \widetilde{Player}\}_{Server}$	Start the game
$Player_i \longrightarrow Server : H_i, \{id, H_i\}_{Player_i}$	Commit move, where $H_i = hash(Player_i, id, M_i)$
$Server \longrightarrow Player_i : \widetilde{H}, \{id, \widetilde{H}\}_{Player}, \{id, \widetilde{H}\}_{Server}$	Commit list
$Player_i \longrightarrow Server : M_i$	Reveal move
$Server \longrightarrow Player_i : \widetilde{M}$	Reveal list

Each player first contacts the game server (Hello). Once a party of n players is ready, the server informs the players that the game starts (Start): it generates a fresh game identifier id and signs it together with the list of players $\widetilde{A_i}$ for the game.

After accepting the server message, each player selects a move M_i and commits to it (Commit move): he computes and signs the hash of his move together with the game identifier (to prevent replay attacks) and his own name (to prevent reflection attacks). The server countersigns and forwards all commitments to all players (Commit list).

After accepting the server message and checking all commitments, each player unveils his move to the server (Reveal move). The server finally publishes all moves (Reveal list), hence any party can now compute the outcome of the game.

Prevented attacks

- *Message integrity and secrecy attacks* Malicious principals control the environment: any of the messages sent by the participants are public and may be corrupted on the way to their destination. Cryptographic hashing of a principal's move prevents the opponent from discovering the move before the end of the bidding phase. Cryptographic signatures allow to detect corruption of a message, but also authenticate the signer.
- *Replay attack* Both the server and the players include the game identifier, chosen fresh at protocol start, in their signatures. Without this, for instance, once a malicious player knows both the sealed and the revealed move of an honest player, he could attempt repeating it in another game. The game identifier binds each cryptographic message element to a particular game.
- *Reflection attack* Players embed their identity inside their sealed move. Without this, a malicious player collaborating with the server could "copy" the moves of an honest player, and so share the gain in case of victory. For this he should simply wait for an honest player to issue his sealed move, countersign the hash (without even knowing it), and use it as its own sealed move. After the end of the bidding phase he should again wait for the honest player to reveal his move, and just copy it.

Protocol implementation The complete, verified source code for the protocol implementation appears online [cod]. It consists of 280 lines of F7 declarations and 420 lines of F# definitions, excluding the standard F7/F# libraries for networking and cryptography. The code is reasonably complex, partly because of the tension between confidentiality and authentication/auditability, partly because it supports any number of players. In particular, the protocol operates on signed lists with embedded hashes, and one could easily miss one step of their validation. (In the process of prototyping, we identified and fixed several errors involving ambiguous message formats.) Automated verification for n-ary group protocols and their implementations is still largely an open problem, even for confidentiality and authentication [Baughman and Levine, 2001].

Security goals (informally) Our protocol offers several properties.

- *Integrity:* the messages (Start), (Commit) and (Commit list) are authenticated.
- *Secrecy:* each player's move remains secret until successful completion of the commitment round, hence the other players' moves for this game cannot depend on it.
- *Auditability:* once a player wins a game id, it can reliably convince all other principals of his victory (according to a "judge" procedure, defined below).

To prove his wins, each player collects the verified commitments from the other players, as well as the second server signature. We now explain what constitutes evidence for this property, first operationally, by defining our judge function, then from a specification viewpoint, by defining formulas that relate the actions of the participants.

Judge and evidence Our target property is *Wins*(*server*,*id*,*players*,*winner*,*move*), a predicate parameterized by the server principal, the game identifier, the list of players, the winner principal, and the winning move. Below we list the judge, as defined in our ML implementation: a function that takes the same parameters plus some evidence $e = (ssig2, evl)$:

```
let judgeWins server n winner move e =
  let (ssig2,evl) = e in
  let players,(hashes:hash list),moves,sigs = unzip4 evl in (* (1) *)
  let vk = getPublicKey usage server in
  let b1 = payload2bytes (CommitList_data(n,players,hashes)) in
  if rsa_verify_prin usage server vk b1 ssig2 then (* (2) *)
  if forall_hash verify_hash n evl then (* (3) *)
  assert(ValidHashes(n,evl));
  if forall_move verify_move n evl then (* (4) *)
  assert(ValidMoves(n,evl));
  if winning_move move moves then (* (5) *)
  if exists winner move evl then (* (6) *)
  true
  else false (* not actually played *)
```

and that calls the two auxiliary functions

```
let verify_hash n (player,hash,move,sg) = (* (3) *)
  let vk = getPublicKey usage player in
  let b = payload2bytes (Commit_data(n,hash)) in
  rsa_verify_prin usage player vk b sg

let verify_move n (player,hash,move,sg) = (* (4) *)
  let b = move2bytes (player,n,move) in
  let h' = sha1 b in
  hash = h'
```

Function *verify_hash* checks a blinded move against its signature for a given principal. Function *verify_move* checks a blinded move against the real move.

Our implementation relies on the *Principals* library for key management. The source code of the game refers directly to the principals' identities and call the corresponding key retrieval functions before generating or verifying a signature. For instance, the judge retrieves the public keys of the server and, using the iteration on the list, of all the players to verify their signatures. The variable *usage* globally identifies the keys used by this protocol in the key database.

Functions *payload2bytes* and *move2bytes* marshal the protocol messages and the players' moves, respectively. Moves are blinded using the SHA1 hash function.

The evidence should consist of the server's signature on the committed hashes (*ssig2*) and a list (*evl*) of 4-tuples $A_i, H_i, M_i, \{N, H_i\}_{A_i}$ (one for each player). This evidence is checked as follows:

(1) split the tuple list into 4 lists of the respective tuple components, using a variant of the ML library function *List.unzip*; then check that

(2) the server's signature on the hashes is valid;

(3) for each 4-tuple, the hash is well-signed;

(4) for each 4-tuple, the hash is correctly computed from the move;

(5) *move* meets some game-specific victory condition; and

(6) *winner* actually played this *move*.

Finally, return **true** if all those checks succeed, **false** otherwise. The code uses monomorphic variants of the ML library function *List.forall* that calls a boolean function on each element of a list and returns **true** if all those calls return **true**. The code of this function is replicated as *forall_hash* and *forall_move*, and given call-site specific types needed for typechecking. In

Chapter 6 we work around this limitation of F7 so that code replication is not needed any more.

Logical Properties To convince ourselves (and the players) that our judge is indeed correct, and that our player is auditable for *Wins*(*s*,*id*,*pls*,*w*,*m*), we now associate logical properties with each message, at each point of the protocol. This association is enforced by typechecking our code against refinement types that embed these properties. Thus, these properties form the basis for our security verification. We refer to the code for their complete, formal definition. By convention, when a property can be attributed to a principal, the corresponding predicate records that principal as its first argument. We first specify the events assumed by the principals before signing. To sign a message, the corresponding predicate must be assumed.

Message	Assumption	Meaning
Start	$Start(s, players, id)$	server s started game id with $players$
Commit	$Commits(p, id, hash)$	player p committed to $hash$ in game id
Commit list	$CommitList(s, id, hashes)$	server s collected $hashes$ in game id

We also define auxiliary predicates for verifying our code, for instance recursive predicates on lists. These definitions often provide convenient abbreviations which hide the details of the logical formulas attached to cryptographic constructs. *ValidHashes* holds for some evidence when all the players committed to their hashes *ValidMoves* holds when all the players the moves of all the players match their blinded moves.

Predicate *Mem* defines list membership. Predicate *Zip4*(l,l_1,l_2,l_3,l_4) is the verified post-condition of a function *unzip4* that splits a list l of 4-tuples into four lists l_1, l_2, l_3, and l_4. Predicate $Winning(m, ms)$ holds when the function $winning\_move(m, ms)$ returns true.

The main rule of the game puts all these pieces together, formalising when the players and the server concede victory, as an assumption that defines the *Wins* predicate:

$$
\begin{aligned}
&\forall server,n,winner,move.\\
&\forall players,moves,evl,r1,hashes,sigs,vks.\\
&\quad (Zip4(evl,players,hashes,moves,sigs)\\
&\quad \wedge (Game(server,players,n) \wedge CommitList(server,n,hashes)) \vee Bad(server)\\
&\quad \wedge ValidHashes(n,evl) \wedge ValidMoves(n,evl)\\
&\quad \wedge Winning(move,moves)\\
&\quad \wedge \exists hash,sg.\ Mem((winner,hash,move,sg),evl)\\
&\Rightarrow Wins(server,n,winner,move)
\end{aligned}
$$

Victory is inferred when *server* started a game *id* for some *players* (*Start*) and collected some commitments *hashes* (*CommitList*), and when there are *moves*, and *sigs* that form a list of evidence *evl* (*Zip4*), such that (i) (sanity checks) in each tuple, the hash is obtained from the move (*ValidMoves*), and the principal signed his hash (*ValidHashes*); (ii) (victory check) *move* is the best move among all *moves* (*Winning*), and *winner* did play *move* (*Mem*).

Principals and Partial Compromise Our model finally accounts for compromised players and servers; to this end, we provide a public interface for creating both good and bad (compromised) principals. Before releasing a signing key to the opponent, to satisfy the pre-condition of the key compromise library function *Principals.rsa_keycomp* (see Section 2.4.4) we formally assume *Leak*(p), which collects any assumption that the compromised principal p may ever make, both as a server and as player:

assume $(\forall p.\ Leak(p) \Rightarrow (\forall n,x.\ Game(p,x,n) \wedge CommitList(p,n,x) \wedge Commits(p,n,x)\ )\ )$

Player (with an Audit statement) In contrast with the code for the judge, the players need not agree on the code for the server and the other players. Still, a player willing to use our client code may wish to review when this code performs actions on his behalf (relying on

the **assert**s statements), and when this code has enough evidence to prove his wins (relying on the **audit** statement). The code for the player is listed in Appendix E.1. The **audit** statement appears after successfully processing all three messages from the server. The gathered evidence consists of the server's signature for the list of commitments, and the list of 5-tuples representing all moves.

Security (formally) We can now precisely state and prove our security goals. The most interesting result is that, for any number of games between any number of players, for any assignment of the server and these players to principals, any player's win is auditable, even if all other participants are corrupted and collude against this player.

Theorem 5.3 (Security of the n-players game)**.** *Let* $\langle \mathcal{L}, I_{\mathcal{L}} \rangle$ *match the protocol obtained by composing the Net,Db,Crypto, and Principals libraries and the source code* $\mathcal{L}_{game}$ *of our game protocol with the type interface* $I_{\mathcal{L}}$ *of our game protocol.*

(1) *integrity:* $\mathcal{L}$ *is robustly safe;*

(2) *auditability:* $\mathcal{L}$ *is auditable for the property Wins(server,id,players,winner,move);*

(3) *secrecy:* $\mathcal{L}$ *preserves secrecy of the moves until all players commit.*

PROOF: (1) By typing the code and Theorem 2.2, all assertions are always satisfied.

(2) By typing, Theorem 5.1, and a termination argument for the *judge*: its code is a sequence of let bindings on expressions that terminate in linear time in the size of their list parameters, so by construction the *judge* function terminates on all inputs. By typing we have that our protocol code $((Net, Db, Crypto, Principals), \mathcal{L}_{game}, I_{\mathcal{L}})$ is a refined module, and so its composition with the refined library modules is also a refined module.

(3) By typechecking a variant of the code (done for the previous version of F7 cryptographic libraries). We model a move as a function with the *ReleaseMove*(*player*,*id*) precondition (defined below) so that one cannot actually apply the function without satisfying the precondition. Player p may release his move in game id, if a server committed to a list of valid sealed bids for all players including p.

□

Other auditable properties In this protocol if honest players never communicate with each other, a server could run several sessions with subsets of players after receiving their sealed bids, and thus convince several of them of their victory. This attack is similar to multiple commitment in Section 3.2. However this would require him to issue several *CommitList* signatures. The property "server s cheated in a game i" can be audited. To condemn the server it would suffice to find two signatures issued by the server during the same game (identified with the same nonce) for different lists of players and sealed moves. A correct judge would check the well-formedness and compare the signatures. This would allow us to treat corruption in a uniform way at the logic level.

5.3 Related work on checking audit-related properties

Aura [Jia et al., 2008, Vaughan et al., 2008] is a programming language that embeds an authorization logic and automatic audit logging. Its authorization logic, based on a dependently-typed variant Dependency Core Calculus [Abadi et al., 1999], can express various security policies. In particular, the "A **says** C" operator models permission and delegation policies endorsed by principals. Untrusted but well-typed applications willing to access a protected resource, must possess a proof that this access may be granted. Aura enforces logging of these

proofs as evidence for *a posteriori* audit. Compared to the F7 typechecker, which uses formulas only for typechecking then erases them, Aura's logic constructs and proofs are first-class citizens, computed and manipulated at runtime. Aura has no specific support for cryptography: generic signatures of propositions rather than of data terms are allowed, and, relying on signed proof terms, Aura can log these as evidence of any past run. Since, in their design, all authorisations decisions are implicitly auditable, at run-time Aura must carry all generated proof terms (at least before compiler optimizations). In our approach, the programmer exports the terms that will constitute the evidence, as an important, explicit part of the protocol design. The typechecker statically guarantees completeness of the evidence and, at run-time, the judge validates the associated proofs only on demand.

The audit-based logical framework for discretionary access control developed by Cederquist et al. [2007], also discussed in Section 4.5, is not implemented for a particular programming language and does not provide a static analysis method to verify accountability.

Chapter 6

Using pre- and post-conditions to verify auditability

The contribution of this chapter is twofold. We extend the F7 typechecker to support modular verification of higher-order security APIs and compact type annotations for checking auditability.

6.1 Towards more flexibility for F7

In practice, protocol implementations involve various data structures, and thus the need for type annotations extends to various library functions that manipulate this data. Although F7 supports polymorphism à la ML, it is difficult to give these library functions precise, yet polymorphic refinement types. In particular, recursive data processing involves higher-order functions, and the programmer must often provide a refinement type each time he uses these functions. Pragmatically, this involves replicating the code for these functions (and some of the functions they call); annotating each replica with its *ad hoc* type; and letting F7 typecheck the replica for each particular usage. For instance, the code of *List.forall* has to be replicated at different call sites in the multi-party protocol of Section 5.

As another example, when the message format used by a protocol is under development it may change often. Each change trickles down the protocol data flow, demanding many changes to logical annotations, and possibly further code replication. This hinders code modularity. Can we write less code and annotations, and focus on the security properties of our program? In this chapter we show how using automatic predicates for pre- and post-conditions allows us to write more flexible and reusable types.

Example Consider the type $\alpha\ option$, which is part of the standard ML library. Its instance $int\ option$ is the type of optional integers: its values range over $None$ and $Some\ n$, where n is an integer. Using option types, we can, for example, program protocols that have optional fields in their messages. To manipulate a message field of type $int\ option$, it is convenient to use the higher-order library function *map*:

val *map*: $(int \to int) \to int\ option \to int\ option$
let *map* $f\ x =$ **match** x **with**
| $None \to None$
| $Some(v) \to$ **let** $w = f\ v$ **in** $Some(w)$

This function can be applied to any function whose type is a subtype of $int \to int$, of the form $x{:}int \to y{:}int\{C(x,y)\}$ for some formula C that can refer to both x and y. Suppose that we compute a value using *map* over a function f with type $v : int \to w : int\ \{w{>}v\}$:

let $y = map\ f\ (Some(0))$

We would like to give y a type that records the post-condition of f:

val $y{:}int\ option\{\exists w.\ y = Some(w) \wedge w > 0\}$

What type must *map* have in order for y to have this type? Within RCF, the most precise type we can give is

val $map{:}\ f{:}(int \to int) \to x{:}(int\ option) \to y{:}(int\ option)$
$\{\ (x = None \wedge y = None) \vee (\exists v,w.\ x = Some(v) \wedge y = Some(w))\}$

This type accounts for the various cases (*None* vs *Some*) of the argument, but not for the post-condition of f. In RCF, terms in formulas range over ML values, such as $Some(w)$, but not expressions, such as $f\ x$, since their evaluation may cause and depend on side-effects. Thus, the only way to check that y has its desired type is to copy the definition of the *map* function just for f and to annotate and typecheck it again:

val $map\_copy{:}\ f{:}(v{:}int \to w{:}int\ \{w{>}v\}) \to x{:}(int\ option) \to y{:}(int\ option)$
$\{\ (x = None \wedge y = None) \vee (\exists v,w.\ x = Some(v) \wedge y = Some(w) \wedge w > v)\}$

Our main idea is to let the F7 typechecker automatically inject and check annotations for pre- and post-conditions. This yields precise, generic types for higher-order functions, thereby preventing the need for manual code duplication and annotation. To this end, we introduce predicates *Pre* and *Post* within the types of higher-order functions to refer to the pre- and post-conditions of their functional arguments. For instance, the formula $Post(f,v,w)$ can refer to the post-condition of a function parameter f applied to v returning w, and we can give *map* the type:

val $map{:}\ f{:}(int \to int) \to x{:}int\ option \to y{:}int\ option$
$\{(x = None \wedge y = None) \vee (\exists v,w.\ x = Some(v) \wedge y = Some(w) \wedge Post(f,v,w))\}$

Whenever *map* is called (say within the definition of y above), the actual post-condition of f is statically known ($w{>}v$) and can be used instead of $Post(f,v,w)$. Hence, y can be given its desired type without loss of modularity.

More generally, we show how to use *Pre* and *Post* predicates to give precise reusable types to a library of recursive higher-order functions for list processing, and use the library to verify protocol implementations using lists. Verifying such implementations is beyond the reach of typical security verification tools, since their proof requires some form of induction. For example, Fs2pv [Bhargavan et al., 2008b] compiles F# into the applied π-calculus, for analysis with ProVerif [Blanchet, 2001], a state-of-the-art domain-specific prover. Although Fs2pv and ProVerif are able to prove complex XML-based cryptographic protocol code, they do so by bounding lists to a constant length and then inlining and re-verifying the list processing code at each call site.

6.2 Refinements for pre- and post-conditions

Classically, for a given function application, a pair of formulas (C_1, C_2) is a valid pair of pre- and post-conditions when, if C_1 holds just before calling the function, then C_2 holds just after the function completes. Hoare [1969] originally proposed them for arbitrary programs. More recently, for example, Spec# [Barnett et al., 2005] and Code Contracts [Fähndrich et al., 2010] let function definitions be annotated with contracts (formulas) expressing intended pre- and post-conditions, which may be checked statically or dynamically. F7 naturally supports pre- and post-conditions for functions as refinements of their argument and return types. For instance, if an F7 function has type $x_1 : P_1\{C_1\} \to x_2 : P_2\{C_2\}$, then asserting C_1 before the function call and C_2 after the function returns is always safe.

In this section we show how to explicitly refer to pre- and post-conditions of functions using generic predicates indexed by function value. There are at least three ways to define the semantics of these predicates. When considering a program with verification annotations, the pre- and post-condition of a function can refer either to the formulas declared with that function, or to the formulas available at the call site, or to events tracking run-time calls and returns. For each semantics, we introduce a pair of generic predicates, informally explain their use, and then give (1) a formal code transformation; and (2) a patch to the F7 typing rules to implement and validate this semantics.

6.2.1 Event-based semantics

Pre- and post-conditions can be seen as events marking the beginning and the end of a function execution. We systematically record them by assuming facts for two predicates *Call* and *Return*: the fact *Call*(M,N) means that M is a function that has been applied to the argument N; *Return*(M,N,O) means that M is a function that has been applied to N and has returned the value O. Formally, this yields a concrete, extensional, finite model, for each partial run of a whole program.

We can use *Call* and *Return* to reason about run-time events, instead of introducing *ad hoc* predicates for that purpose. For instance, if a function *send* parameterized by m assumes a "begin event" *Send*(m) before signing a message with payload m, we can remove this assume and use instead the generic event *Call*(*send*,m) in security specifications. Similarly, suppose that keys are represented as bitstrings, but that the keys in use should be generated only by a designated algorithm *genKey*. We can assign to keys the refinement type k : *bytes* { *Return* (*genKey*,(),k) }. This pattern frequently applies to cryptographic materials such as nonces, initialization vectors, and tags.

To preserve consistency of the assumed formulas, we rely on a standard notion of positive and negative positions in types and formulas. (Hence, for instance, within the formula $\forall b1$,$b2$,c. *Call*(*concat*,[$b1$;$b2$],c)$\Rightarrow$ *IsConcat*(c,$b1$,$b2$) the predicate *Call* occurs negatively while the predicate *IsConcat* occurs positively.) In the program before the transformation, we forbid positive occurrences of *Call* and *Return* in assumed formulas.

Code transformation We specify the event-based semantics by translating every syntactic function and every function application

$$\begin{aligned} [\![\ \textbf{rec}\ f\colon T.\ \textbf{fun}\ x \to e]\!]_E &\triangleq \textbf{rec}\ f\colon T.\ \textbf{fun}\ x \to \textbf{assume}\ \mathit{Call}(f,x); [\![e]\!]_E \\ [\![\ M\ N]\!]_E &\triangleq\ \textbf{let}\ r = [\![M]\!]_E\ [\![N]\!]_E \textbf{in assume}\ \mathit{Return}(M,N,r);\ r \end{aligned}$$

(where r is fresh in M,T, and N) and letting $[\![]\!]_E$ be a homomorphism for all other expressions. Thus, we bracket each call with events before and after the call.

Modifying the typechecker We achieve the same effect as the transformation by directly injecting formulas when typechecking functions and applications. We replace the typing rule (Typ Fun) with (Typ Fun PrePost), and the typing rule(Typ App) with (Typ App PrePost), as shown below. We call the resulting type system RCF_E.

RCF

$$\text{(Typ Fun)}\quad \frac{E \vdash x : T_1 \to T_2 <: T \qquad E, f : T, x : T_1 \vdash e : T_2}{E \vdash \mathbf{rec}\ f : T.(\mathbf{fun}\ x \to e)\ :\ x : T_1 \to T_2}$$

$$\text{(Typ App)}\quad \frac{E \vdash M : x : T_1 \to T_2 \qquad E \vdash N : T_1}{E \vdash (M\,N) : T_2}$$

RCF$_E$

$$\text{(Typ Fun PrePost)}\quad \frac{T = x : T_1 \to T_2 \qquad E, f : T, x : T_1, \mathit{Call}(f, x) \vdash e : T_2}{E \vdash (\mathbf{rec}\ f : T.\mathbf{fun}\ x \to e) : T}$$

$$\text{(Typ App PrePost)}\quad \frac{E \vdash M : x : T_1 \to (r : P)\{C\} \qquad E \vdash N : T_1 \qquad T_2 = (r : P)\{C \wedge \mathit{Return}(M, x, r)\}}{E \vdash (M\,N) : T_2\{N/x\}}$$

Results We check that our transformation does not affect the operational behaviour, safety, and well-typedness of programs that do not use *Call* and *Return*, and that the code transformation and the modified typing rules yield the same typing judgements. Let e be a closed program. Let $e \Downarrow M$ denote evaluation of the expression e ($e \longrightarrow^* \nu\tilde{a}.e' \Rsh M$ where e' consists of assumptions and auxiliary threads).

Lemma 6.1. *Suppose that Call and Return do not occur in e.*
- Evaluation: *for any value* M, $e \Downarrow M$ *if and only if* $[\![e]\!]_E \Downarrow [\![M]\!]_E$;
- Safety: e *is safe if and only if* $[\![e]\!]_E$ *is safe; and*
- Typing: e *is well-typed in RCF if and only if* $[\![e]\!]_E$ *is well-typed in RCF.*

Lemma 6.2. $[\![e]\!]_E$ *is well-typed in RCF if and only if* e *is well-typed in RCF*$_E$.

The proofs of these lemmas can be found in Appendix D.

6.2.2 Macro-expansion semantics

Pre- and post-conditions may also be seen as pure syntactic sugar, abbreviations that refer to concrete formulas in the types of functions in scope (similar to the *pre* and *post* projections of Régis-Gianas and Pottier [2008]). It is useful to refer to the pre- or post-condition of a known and fully annotated function to avoid copying a formula which is big or likely to change during the verification process.

To denote such macro-definitions, we introduce generic predicates $\#\mathit{Pre}$ and $\#\mathit{Post}$. They may occur anywhere in the program or its interface, provided that their first argument is a variable name that has a declared function type in their scope. *Before typechecking,* we replace each of their occurrences with a concrete formula read off the environment without breaking well-formedness.

Implementation If $E(f) = x_1 : P_1\{C_1\} \to x_2 : P_2\{C_2\}$, then we replace $\#\mathit{Pre}(f, M)$ with $C_1[M/x_1]$, and $\#\mathit{Post}(f, M, N)$ with $C_2[M/x_1, N/x_2]$. If the lookup fails, or the returned type is not a function type, preprocessing fails—the macro-definition is ill-formed.

6.2.3 Subtyping-based semantics

As opposed to the type annotations of function definitions, the declared types of function arguments in higher-order functions are in general only supertypes of the argument types actually used at their call sites, themselves supertypes of the functional types verified at the function definitions. Thus, as we type the higher-order function, the actual refinements for its argument are unknown, and we cannot just rely on macro-expansion. We refer to these refinements using predicates *Pre* and *Post*.

- We use them parametrically when typing higher-order functions, as if each function argument f had a type of the form $x_1 : P_1\{\mathit{Pre}(f, x_1)\} \to x_2 : P_2\{\mathit{Post}(f, x_1, x_2)\}$.

– We define their logical model as follows: for each closed function value, of the form

$$M = \textbf{rec}\ f : (x_1 : P_1^\circ)\{C_1^\circ\} \to (x_2 : P_2^\circ)\{C_2^\circ\}.\textbf{fun}\ x_1 \to e$$

  – $\mathit{Pre}(M,M_1)$ if and only if $C_1^\circ[M_1/x_1]$ and
  – $\mathit{Post}(M,M_1,M_2)$ if and only if $C_1^\circ[M_1/x_1] \Rightarrow C_2^\circ[M_1/x_1, M_2/x_2]$.
– When applying a function parameter f of type $T = x_1{:}P_1\{C_1\} \to x_2{:}P_2\{C_2\}$ at the call site, for any runtime instance M of f of the form above, we have $\forall x_1.\ C_1 \Rightarrow C_1^\circ$ and $\forall x_1,x_2.\ C_1 \wedge C_2^\circ \Rightarrow C_2$ by type safety and subtyping. Accordingly, for relating C_1 and C_2 to the (unknown) parametric pre- and post-conditions of f during typechecking, we automatically assume the formula

$$\phi_{f:T} = \forall x_1.\ C_1 \Rightarrow \mathit{Pre}(f,x_1) \wedge \forall x_1,x_2.\ (C_1 \wedge \mathit{Post}(f,x_1,x_2)) \Rightarrow C_2$$

  Intuitively, we treat f as if it had the refinement type

$$f{:}(x_1{:}P_1\{\mathit{Pre}(f,x_1)\} \to x_2{:}P_2\{\mathit{Post}(f,x_1,x_2)\})\ \{\ \phi_{f:T}\ \}$$

  which is a subtype of $f{:}(x_1{:}P_1\{\mathit{Pre}(f,x_1)\} \to x_2{:}P_2\{\mathit{Post}(f,x_1,x_2)\})$.

Relation to the event-based semantics Within the body of a higher-order function with function argument f, whenever f is applied to a value N, the event $\mathit{Call}(f,N)$ records this application, and typing requires that the predicate $\mathit{Pre}(f,N)$ holds. At runtime, for each instance M of f, the actual pre-condition of M holds (by typing) and implies the formal precondition of f (by assumption) so we have

$\forall f,x.\ \mathit{Call}(f,x) \Rightarrow \mathit{Pre}(f,x)$

Similarly, when f returns, we have $\mathit{Return}(f,N,O)$, and its formal post-condition $\mathit{Post}(f,N,O)$ implies the actual post-condition for any instance M of f (by assumption) so we have

$\forall f,x,y.\ \mathit{Return}(f,x,y) \Rightarrow \mathit{Post}(f,x,y)$

We thus assume both of these formulas for typechecking.

Code transformation To support *Pre* and *Post*, we rely on events, so we first apply the event-based code transformation, then we transform every binding whose expression has a function type annotation and apply $[\![]\!]_S$ homomorphically to other expressions. In particular, we transform every function let binding (since they are always annotated) and every syntactic function definition, ensuring that all functional arguments are annotated in higher-order functions. Let T abbreviate $x_1{:}P_1\{C_1\} \to x_2{:}P_2\{C_2\}$.

$$[\![\ \textbf{let}\ f = e : (f : T\ )\ \{C_f\})\ \textbf{in}\ e'\ ]\!]_S \triangleq \textbf{let}\ f = [\![e]\!]_S\ \textbf{in assume}\ \phi_{f:T};\ [\![e']\!]_S$$
$$[\![\ \textbf{rec}\ f{:}\ T.\ \textbf{fun}\ x \to\ e\ ]\!]_S \triangleq \textbf{rec}\ f{:}\ T.\ \textbf{fun}\ x \to [\![\textbf{let}\ x = (x : T_1)\ \textbf{in}\ e]\!]_S$$

Modifying the typechecker We modify F7 to support *Pre* and *Post* by modifying insertions of variables entries with function types into the typing environment. Hence, E extended with $f : T$ is now written $E \oplus f : T$, and defined by pattern matching on T. If T is a function type, it is of the form $f{:}(x_1{:}P_1\{C_1\} \to x_2{:}P_2\{C_2\})\{C_f\}$ and we let

$$E \oplus f : T \triangleq E, f : T, \phi_{f:(x_1:P_1)\{C_1\}\to(x_2:P_2)\{C_2\}}$$

Otherwise $E \oplus f : T$ is just $E, f : T$. We call the modified type system RCF_S. To maintain logical consistency, we forbid positive occurrences of *Pre* and *Post* in assumed formulas.

Results

We obtain a variant of Lemma 6.1 for the subtyping semantics: we have a similar Evaluation property. The proof of Safety involves showing the logical consistency of the injected assumptions.

Lemma 6.3. *Suppose that Call and Return do not occur in e.*
- Evaluation: *for any value M, e* $\Downarrow$ *M if and only if* $[\![e]\!]_S \Downarrow [\![M]\!]_S$;
- Safety: *e is safe if and only if* $[\![e]\!]_S$ *is safe; and*

We also prove two flavours of Correctness. We have a variant of Lemma 6.2 that relates typing with RCF_S and the specification $[\![]\!]_S$.

Lemma 6.4. $[\![e]\!]_S$ *is well-typed in RCF if and only if e is well-typed in* RCF_S.

Besides, we show a simple pattern such that *Pre* and *Post* can be eliminated by replicating the code of a higher-order functions at each call site and annotating each replica with an *ad hoc* type.

Theorem 6.5 (Inlining)**.** *Let* $e_0 =$ ***let*** $h = H$ ***in*** e *be a well-typed expression in* RCF_S *such that H is a function with type* $T = g{:}(x{:}T_1 \to T_2) \to T_3$*, h only occurs in applicative position and Pre and Post occur only in* T_3 *and always have g as first argument.*

Let e_1 *be* e_0 *after replacing each subexpression of the form* $(h\ f) : T'$ *in e with* $(H_f\ f) : T'$*, where* H_f *is H after replacing each Pre(g,M) and Post(g,M,N) with #Pre(f,M) and #Post(f, M,N), respectively.*

Then e_1 *is also well-typed in* RCF_S.

The proofs of these results can be found in Appendix D.

Functions with multiple arguments Our definitions above assume curried functions. For convenience, we have also implemented typechecking support for functions with multiple arguments, recorded as a list in our predicates. For example, the function call *M a b* assumes the event *Call*(*M*,[*a*;*b*]).

6.3 Reusable typed interface for lists

Lists are perhaps the most commonly-used data structures in functional programs. The F# *List* library provides efficient implementations of recursive list processing functions; for generality, these functions are typically higher-order and polymorphic. Our goal is to give this library a reusable refinement typed interface, using our *Pre* and *Post* predicates and their subtyping-based semantics. The full interface is listed in Appendix E.2.

We detail our approach on the function *List.fold*, the general iterator on lists (also called *fold_left*). Its ML type is $(\alpha \to \beta \to \alpha) \to \alpha \to \beta\ list \to \alpha$. It takes as argument a function *f*, an initial *accumulator a*, a list *l* and traverses the list *l*, applying *f* to the current accumulator and the next value in the list to obtain the next accumulator; when it reaches the end of the list, it returns the accumulator. For example, *fold* (+) 0 [1;2;3;4] computes the sum of the elements in the list.

To compute the sum of elements of a list *l* of integers using function (+): $x{:}int \to y{:}int \to z : int\{z{=}x{+}y\}$, we write *fold* (+)0 *l* which can only be given type *int*, whereas we expect it to have a refinement equivalent to $s{:}int\{\sum_{x \in l} x\}$. To achieve this in RCF, we can duplicate the code of *fold* and giving this copy an *ad hoc* type. In this section we show how to give *List.fold* a precise and reusable type that logically relates the iterated function with the accumulator and the result.

First attempt: using recursive predicates Let us define two predicates *PreFold* and *PostFold* to represent the pre- and post-condition of *fold*. By inspecting the code for *fold* (on the left below) we can define these predicates as shown:

```
let rec                          assume ∀f,acc,l.                 assume ∀f,acc,l,r.
  fold f acc l =                 PreFold(f,acc,l)                 PostFold(f,acc,l,r)
  match l with                   ⇔                                ⇔
   | [] → acc                     (l=[]                            ((l=[] ∧ r=acc)
                                  ∨                                ∨
   | hd :: tl →                   (∃hd,tl. l=hd::tl ∧              (∃hd,tl. l=hd::tl ∧
     let acc' = f acc hd in       Pre(f,[acc;hd]) ∧
                                  (∀acc'. Post(f,[acc;hd],acc')    (∃acc'. Post(f,[acc;hd],acc')
     fold f acc' tl                     ⇒ PreFold(f,acc',tl))))    ∧ PostFold(f,acc',tl,r))))
```

The definition for *PreFold* can be read as follows. If the list is empty, there is no pre-condition. Otherwise, the pre-condition of the argument f must hold for the head of the list and the current accumulator, and if f terminates and returns a new accumulator, *PreFold* must hold for the tail of the list and this new accumulator. The post-condition *PostFold*, if the list is not empty, is that the iterated function applied to the head of the list and the accumulator must return a new accumulator so that *fold* applied to this new accumulator and the tail of the list returns this value.

The resulting type for *fold*

$$\textbf{val}\ fold\colon f\colon(\alpha \to \beta \to \alpha) \to acc\colon\alpha \to l\colon\beta\ \ list\{PreFold(f,acc,l)\} \to r\colon\alpha\ \{PostFold(f,acc,l,r)\}$$

is precise and easy to typecheck against the code of *fold*, yet difficult to use at call sites. Indeed, even for a function with no pre-condition ($\forall x.\ Pre(f,x)$), proving $PreFold(f,acc,l)$ requires the use of induction, which is generally beyond the reach of the SMT solver Z3 that underlies F7. Can we use a non-recursive predicate to specify *fold*?

Second attempt: using invariants In our second approach, we adopt the style of Régis-Gianas and Pottier [2008] for specifying higher-order iterators, such as *fold*. We introduce a generic predicate *Inv* that is used to define logical *invariants* for functions that may be used as an argument to *fold*. The formula $Inv(f,aux,acc,l)$ is an invariant that holds when the function f is being applied to a list of elements: l is the remainder of the list, acc is the intermediate result of the computation, and aux contains function-specific auxiliary information about the initial arguments to the *fold*.

As an example, consider the function *fmem* that can be used with *fold* to search for an element in a list; its code, refinement type, and invariant are as follows:

```
let fmem v                       val fmem: v:α →                  (∀v,f. Post(fmem,v,f) ⇒
    acc                            acc:bool →                      (∀iv,acc,l. Inv(f,iv,acc,l) ⇔
    n                              n:β →                           (∃x,linit. iv=(x,linit) ∧ x = v
    =                              found:bool{                     ∧ (∀y. Mem(y,l) ⇒ Mem(y,linit))
  if v = n                         (v = n                          ∧ (( Mem(x,linit)
  then true                          ∧ found = true)                        ∧ acc=true)
  else acc                       ∨ (found = acc)}                       ∨ acc = false ))))
```

The function *fmem* takes an element v to search for, an accumulator *acc* and an integer n, and returns true if either $v = n$ or *acc* is true. The invariant for the function obtained by the partial application *fmem* v is defined on the right; its auxiliary argument *aux* is a pair consisting of the searched element v and the initial list *linit*. Its auxiliary argument is a pair consisting of the integer v to search for, and the initial list *linit*. The invariant says that the remaining list l contains a subset of the elements in *linit*, and that the accumulator is true only if v is a member of *linit*.

The next step, following Régis-Gianas and Pottier, is to prove that the invariant is *hereditary*, namely that the invariant of each function f is at least as strong as its pre-condition, and that the invariant is preserved by function application. We define a predicate *Hereditary* that captures this notion and use it to give a type to *fold* as shown below; note that to use this style we need

to add an additional argument *aux* to *fold*.

```
let rec fold v f acc l =                    assume (∀f. Hereditary(f)
  match l with                                ⇔
    | [] → acc
    | hd :: tl →                              ( ∀v,acc,h,t. Inv(f,v,acc,hd::tl) ⇒
      let acc' = f acc hd in                  (Pre(f,[acc;hd])
      fold v f acc' tl                        ∧ (∀r. Post(f,[acc;hd],r) ⇒ Inv(f,v,r,tl)))))

val fold : v: γ → f:(α → β → α ) {Hereditary(f)} → acc:α
    → xs:β list {Inv(f,v,acc,xs)}
    → r:α { (xs = [] ∧ r=acc) ∨ Inv(f,v,r,[]) }
```

The type of *List.fold* requires that (1) the invariant of the iterated function is hereditary; and that (2) the invariant holds for the initial accumulator. The post-condition states that the invariant holds for the final accumulator.

For example, to check that the application *fold* (*v*,*l*) (*fmem v*) **false** *l* can be given the type *b* :*bool* {*b* = **true** ⇒ *Mem*(*v*,*l*)} we must prove that ∀*v*,*f*. *Post*(*fmem*,*v*,*f*)⇒ *Hereditary*(*f*), and that the invariant of *fmem v* holds for the initial values (**false**,*l*). For a simple function like *fmem* this can be proved automatically, but for more complex functions *Hereditary* may have to be proved by hand. The rest of the typechecking is fully automatic.

6.4 Compact types for audit

Using event-based semantics With the original F7 typechecker the programmer has to define his own auxiliary predicates corresponding to these events and enforce their relationship to the function calls by assuming them within protocol code. In many cases though, she can use the predicates *Call* or *Return* which are declared and managed automatically.

For instance, in the simple authentication example shown in Figure 4.1, we can omit declaring and assuming the predicate *ASent*. Instead we can refer to the automatic event *Call* of the function *princA*. Then we redefine Bob's audit goal as follows:

assume ∀*m*. *AMightSend*(*m*) ⇔ (*Call*(*princA*,[*m*]) ∨ *Pub*(*ska*))

Similarly, in the implementation of the multi-party protocol, the occurrences of predicate *Winning*(*m*,*ms*) can be replaced with *Return*(*winning_move*,[*m*;*ms*],**true**).

Using macro-expansion semantics To formally check the auditability of this protocol, first we show that the judge is *correct*: whenever it returns **true**, the audited property holds. This can be written as the following query:

∀*text*,*e*. #*Post*(*judge*,[*text*;*e*]) ⇒ *Call*(*princA*,[*text*])

Then, to check that at every audit point the judge would have returned **true**, we set the pre-condition of the **audit** primitive to #*Post* of the judge:

val audit: *text*:*bytes* → *e*:*bytes* {#*Post*(*judge*,[*text*;*e*])} → *unit*

The use of the macro-expansion predicate #*Post* here is a convenient way of making this type dependent on the type of judge, hence avoiding the need to rewrite it when the protocol or the judge change. Note that we cannot use the *Return* event or the *Post* predicate here (instead of #*Post*) because both rely on a function having been called; here the call to **audit** must be typeable without actually calling the judge function.

Using subtyping-based semantics Since the number of the players is a run-time parameter of the protocol, the participants have to manipulate lists of cryptographic evidence. For instance, the function *List.forall* is used for three different series of checks, so in the original

F7 implementation the function code had to be replicated and equipped with different *ad hoc* types. Using the subtyping-based semantics, the same library function *List.forall* is used at all call sites, which makes the code more compact and more readable.

6.5 Pre- and post-conditions for protocol implementations

We can use our new types for lists to verify more realistic cryptographic applications. We present two case studies of programs previously verified using F7 and how our extensions help reduce the number of annotations required for typechecking.

6.5.1 XML digital signatures

The XML digital signature standard specifies cryptographic mechanisms to provide integrity, message authentication, and signer authentication for arbitrary XML data [Eastlake et al., 2002]. These mechanisms are used within web services security protocols to protect messages, and processing each message involves tree and list processing. For example, consider a single-message protocol, where the principal a uses an XML signature to protect $n \geq 1$ XML elements $m_1, \ldots, m_n$ located at URIs #1, ..., #n within the message, using the MAC key k_{ab}. The main security goal for this protocol is that the list $[m_1; \cdots; m_n]$ be authenticated; the protocol is often used as a component within a larger protocol that enforces more abstract security properties. The protocol with a slightly simplified message format can be written as follows:

$$
\begin{array}{l}
a \longrightarrow b : \langle \texttt{Message} \rangle \\
\quad m_1\ m_2\ \ldots\ m_n \\
\quad \langle \texttt{Signature} \rangle \\
\quad\ \langle \texttt{SignatureInfo} \rangle \\
\quad\quad \langle \texttt{Reference} \rangle \texttt{base64 (sha1 } m_1\texttt{))} \ \langle \texttt{/Reference} \rangle \\
\quad\quad \cdots \\
\quad\quad \langle \texttt{Reference} \rangle \texttt{base64 (sha1 } m_n\texttt{))} \ \langle \texttt{/Reference} \rangle \\
\quad\ \langle \texttt{/SignatureInfo} \rangle \\
\quad\ \langle \texttt{SignatureValue} \rangle \\
\quad\quad \texttt{base64 (mac } k_{ab}\ (\langle \texttt{SignatureInfo} \rangle \cdots (\textit{as above}) \cdots \langle \texttt{/SignatureInfo} \rangle)) \\
\quad\ \langle \texttt{/SignatureValue} \rangle \\
\quad \langle \texttt{/Signature} \rangle \\
\langle \texttt{/Message} \rangle
\end{array}
$$

In previous work, Bhargavan et al. [2010a] used F7 to program and verify a library for manipulating such XML signatures. Now we can use *List.map* to improve this library and ease its verification. Consider the following excerpt of the library interface and implementation.

```
val mkRef: m:item → r:item{Ref(m,r)}

val xml_sign: a:str → b:str →
  k:key{Return(mkXmlKey,[a;b],k)} →
  ml:item list → dsig:item

val xml_verify: a:str → b:str →
  k:key{Return(mkXmlKey,[a;b],k)} →
  ml:item list → dsig:item →
  unit{Call(xml_sign,[a;b;k;ml])}
```

```
let xml_sign a b k ml =
  let rl = map mkRef ml in
  let si = Xn(signatureInfo,[],rl) in
  let h = hmac k (ditem2bytes si) in
  Xn(signature,[],[si;Xn(sigValue,[],[txt h])])

assume ∀a,b,k. Return(mkKey,[a;b],k) ⇒
  (∀si. MACSays(k,si) ⇔
    (∃ml. Call(xml_sign,[a;b;k;ml]) ∧
      SigInfo(si,ml)))
```

The type *item* represents XML elements; its constructor $Xn(q,al,il)$ corresponds to an XML element of the form $<q\ al>il</q>$, where q is a qualified name, such as `Signature`, al is a list of XML attributes, and il is a list of XML items.

The function *mkRef* generates a *sha1* cryptographic hash of its argument and returns it within a *<Reference>* element. The function *xml_sign* generates an XML signature over a list

of XML elements; it uses *map* over *mkRef* to generate a list of references, encapsulates them within a <*SignatureInfo*> element, and MACs it with the given key *k*. The function *xml_verify* parses and verifies XML signature. Its post-condition guarantees that the signature must have been generated using *xml_sign* by a valid client (as part of an authenticated message).

The use of *List.map* avoids the need to inline the recursive code for *map* in the code for *xml_sign* and *xml_verify*. In our previous verification of the full library, there were four instances where we needed to inline list-processing functions and define new type annotations for each instance. These are no longer necessary, reducing the annotation burden significantly.

6.5.2 X.509 certification paths

The X.509 recommendation [ITU, 1997] defines a standard format and processing procedure for public-key certificates. Each certificate contains at least a principal name, a public-key belonging to that principal, an issuer, and a signature of the certificate using the private key of the issuer.

On receiving a certificate, the recipient first checks that the issuer is a trusted certification authority and then verifies the signature on the certificate before accepting that the given principal has the given public key. To account for situations where the certification authority may not be known to the recipient, the certificate may itself contain a *certification path*: an ordered sequence of public-key certificates that begins with a certificate issued by a trusted certification authority and ends with a certificate for the desired principal. The X.509 sub-protocol for verifying certification paths can be written as follows:

$$\begin{aligned} a \longrightarrow b : \; & \texttt{Certificate}(a_1 \mid pk_{a_1} \mid \texttt{rsa\_sign}\ sk_{CA}\ (a_1 \mid pk_{a_1})) \\ & \texttt{Certificate}(a_2 \mid pk_{a_2} \mid \texttt{rsa\_sign}\ sk_{a_1}\ (a_2 \mid pk_{a_2})) \\ & \ldots \\ & \texttt{Certificate}(a \mid pk_a \mid \texttt{rsa\_sign}\ sk_{a_{n-1}}\ (a \mid pk_a)) \end{aligned}$$

We write and verify a new library for manipulating X.509 certificates. The code for certificate verification uses *List.fold* to iterate through a certification path:

```
val verify:
    x:cert{Certificate(x)} → b:bytes →
    r:cert {Certifies(x,r) ∧ Certificate(r)}
val verify_all:
    x:cert{Certificate(x)} → l: bytes list →
    r:cert {Certifies(x,r)}
```

```
let verify_all ca path =
    fold ca verify ca path

assume ∀ca,x,h,l.
    Inv(verify,ca,x,l) ⇔
      (Certificate(x) ∧ Certifies(ca,x))
```

The predicate *Certifies*(*x*,*y*) specifies that there is some sequence of certificates starting with *x* and ending with *y*, $x{=}x_0, x_1, \ldots, x_n{=}y$, such that the principal mentioned in each x_i has issued the certificate x_{i+1}; hence if every principal mentioned in this sequence is honest, then we can trust that the public-key in the final certificate *y* indeed belongs to the principal mentioned in *y*.

The function *verify_all* takes as argument a certificate *ca* for a trusted certification authority and it accepts only those certification paths that begin with certificates issued with *ca*'s public-key. To typecheck *verify_all* we define the *fold* invariant for *verify* as the property that the accumulator *x* always has a valid certificate (*Certificate*(*x*)) and a valid certification path from *ca* to *x* (*Certifies*(*ca*,*x*)).

The use of *List.fold* in *verify_all* is the natural way of writing this code in ML. We could copy the code for *List.fold* and redo the work of annotating and typechecking it for this protocol, but reusing the types and formulas in *List* is more modular, and we believe, the right way of developing proofs for such cryptographic applications.

6.6 Related work

Pre- and post-condition checking is supported by many program verification tools [e.g. Barnett et al., 2005, Flanagan et al., 2002, Xu, 2006]. Our approach is most closely related to that of Régis-Gianas and Pottier [2008], who show how to use Hoare-style annotations to check programs written in a call-by-value language with recursive higher-order functions and polymorphic types. They extract proof obligations out of programs, and prove them using automated provers. A computational function is logically interpreted as a pair of logical functions binding the argument variables and the result variable to the formulas that serve as pre- and post-conditions in the function type, respectively, like our macro-definitions in Section 6.2.2. A function application $f\,v$ annotated with formula P will for instance generate the proof obligations $pre(f)\,(v)$ and $\forall res.post(f)\,(v)\,(res) \Rightarrow P(res)$. However, their system only uses declared types (#*Pre*,#*Post*), and disregards subtyping and events.

Symbolic methods for verifying the security of protocol implementations utilize a variety of techniques, such as static analysis [Goubault-Larrecq and Parrennes, 2005], model-checking [Chaki and Datta, 2009], and cryptographic theorem-proving [Bhargavan et al., 2008b]. The RCF type system is the first to use refinement types for verifying protocol implementations [Bengtson et al., 2008b]. Its implementation in the F7 typechecker has been successfully used to verify complex cryptographic applications [Backes et al., 2009, Bhargavan et al., 2009, 2010a]. F7 requires programmer intervention in the form of type annotations, whereas some of the other verification tools are fully automated. However, these other tools generally do not apply to programs with recursive data structures. Besides, whole-program analysis techniques seldom scales as well as modular ones, such as typechecking.

Fine [Swamy et al., 2010, Chen et al., 2010] is another extension of F# with refinement types. It also supports affine types and proof-carrying bytecode verification. Its type system has a notion of predicate polymorphism that captures some of the benefits of our pre- and post-condition predicates. To use them, the programmer declares predicate parameters for higher-order types and functions, and explicitly instantiates these predicates at each call site. In contrast, our approach is able to verify legacy programs written purely in F# by automatically injecting pre- and post-condition predicates.

By relying on standard verification techniques, we hope to benefit from their recent progress. For example, Liquid Types [Rondon et al., 2008] have been proposed as a technique for inferring refinement types for ML programs. The types inferred by Liquid Types are quite adequate for verifying simple safety properties of a program, but not for the security types in this paper. We can benefit from such mainstream inference techniques, but some adaptation is required; this is an interesting direction for future work.

6.7 Conclusions and future work

Audit-related properties, like other security properties, require both formal and practical tools for analyzing them. Through Chapters 4 to 6 we showed how to specify and check *auditability* using refined types in F7 in three steps. As a first step, we proposed a general formal definition of auditability as the ability of a program to log enough evidence in order to convince a judge – *a posteriori* – that some property holds. This seemed to be the most general property approximating the "usefulness" of logs as the ability to "pass" an audit.

Crucially, a judge can be modelled as the source code of the function that it runs during the audit and which must be known and accepted by all parties. We can thus build on verification techniques for source code to automatically verify auditability. As a second step, we proposed a type discipline for F7 to express auditability as a type specification for ML code with logical annotations, thus providing a practical analysis tool. Type annotations necessary for refined typechecking with F7 must be provided by the programmer. Guessing the right annotations,

and the success condition for the judge in particular, may not be easy, even if the annotations need not be trusted, and candidate annotations may be "tested" by typechecking.

Finally, we started to investigate how to lighten the annotation burden of programmers. We designed an extension for the F7 typechecker that supports explicitly referring to other functions' pre- and post-conditions. This offers more modularity to the typechecking and often avoids replicating type annotations. As a further step, we plan to develop inference techniques for refined types to reduce the annotations even further.

A future, more challenging, work is to design a tool that compiles audit requirements of the form **audit** C to the minimal complete evidence for a given correct judge. We conjecture that, at least in some cases, the type specification of the judge function carries enough information to enable this synthesis.

Bibliography

Auditability using refined types: code examples. http://msr-inria.inria.fr/~guts/thesis.

Absolute data integrity protection. www.surety.com, 1994.

M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, January 2001. URL citeseer.ist.psu.edu/fournet01mobile.html.

M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–160. ACM, 1999.

M. Abadi, C. Fournet, and G. Gonthier. Secure Implementation of Channel Abstractions. *Information and Computation*, 174(1):37–83, 2002.

P. Adão, C. Fournet, and N. Guts. High-level programming for e-cash. *Workshop on Formal and Computational Cryptography (FCC)*, 2008.

M. Backes, C. Hriţcu, M. Maffei, and T. Tarrach. Type-checking implementations of protocols based on zero-knowledge proofs. In *FCS*, 2009.

M. Barnett, M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, pages 49–69, January 2005.

A. Barth, J. Mitchell, A. Datta, and S. Sundaram. Privacy and utility in business processes. In *Computer Security Foundations Symposium, 2007. CSF'07. 20th IEEE*, pages 279–294. IEEE, 2007.

N. Baughman and B. Levine. Cheat-proof playout for centralized and distributed online games. In *20th Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1, 2001.

M. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 139–154. IEEE, 2004.

M. Bellare and B. Yee. Forward integrity for secure audit logs, 1997.

J. Bengtson, K. Bhargavan, C. Fournet, A. Gordon, and S. Maffeis. Refinement types for secure implementations. Technical Report MSR-TR-2008-118, 2008a.

J. Bengtson, K. Bhargavan, C. Fournet, A. Gordon, and S. Maffeis. Refinement types for secure implementations. In *IEEE Computer Security Foundations Symposium*, pages 17–32, 2008b.

K. Bhargavan, C. Fournet, and A. Gordon. F7: Refinement types for F#. Available at http://research.microsoft.com/en-us/projects/F7/, 2008a. version 1.0.

K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. *ACM TOPLAS*, 31:5:1–5:61, December 2008b.

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

K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *POPL*, pages 445–456, 2010a.

K. Bhargavan, C. Fournet, and N. Guts. Typechecking higher-order security libraries. In *To appear in APLAS*, 2010b.

B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *CSFW*, pages 82–96, 2001.

J. Camenisch, S. Hohenberger, and A. Lysyanskaya. Compact e-cash. In *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 302–321. Springer, 2005.

J. Camenisch, A. Lysyanskaya, and M. Meyerovich. Endorsed e-cash. In *IEEE Symposium on Security and Privacy*, pages 101–115, 2007a.

J. Camenisch, A. Lysyanskaya, and M. Meyerovich. Endorsed e-cash. In *IEEE Symposium on Security and Privacy*, pages 101–115, 2007b.

J. Castellà-Roca, J. Domingo-Ferrer, A. Riera, and J. Borrell. Practical mental poker without a ttp based on homomorphic encryption. In T. Johansson and S. Maitra, editors, *Progress in Cryptology-Indocrypt'2003*, number 2904 in LNCS, pages 280–294. Berlin: Springer-Verlag, 2003. URL citeseer.ist.psu.edu/castella-roca03practical.html.

J. Cederquist, R. Corin, M. Dekker, S. Etalle, J. den Hartog, and G. Lenzini. Audit-based compliance control. *International Journal of Information Security*, 6(2):133–151, 2007.

S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. In *CSF*, pages 172–185, 2009.

D. Chaum. Blind signatures for untraceable payments. In *CRYPTO*, pages 199–203, 1982.

D. Chaum. Secret-ballot receipts : True voter-verifiable elections. *IEEE Security and Privacy*, 2(1):38–47, January/February 2004.

D. Chaum, A. Fiat, and M. Naor. Untraceable electronic cash. In *CRYPTO*, volume 403 of *Lecture Notes in Computer Science*, pages 319–327. Springer, 1988.

D. Chaum, P. Ryan, and S. Schneider. A practical, voter-verifiable election scheme. Technical Report CS-TR-880, 2004.

J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation for end-to-end verification of security enforcement. In *PLDI*, pages 412–423, June 2010.

R. Corin, D. Galindo, and J. Hoepman. Securing Data Accountability in Decentralized Systems. *LNCS*, 4277, 2006.

R. Corin, P.-M. Denielou, C. Fournet, K. Bhargavan, and J. Leifer. Secure implementations for typed session abstractions. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 170–186, 2007.

D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

D. Eastlake, J. Reagle, D. Solo, M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon. *XML-Signature Syntax and Processing*, 2002. W3C Recommendation.

S. Etalle and W. Winsborough. A posteriori compliance control. In *SACMAT*, pages 11–20. ACM Press New York, 2007.

S. Etalle, F. Massacci, and A. Yautsiukhin. The meaning of logs. *Trust, Privacy and Security in Digital Business*, pages 145–154, 2007.

M. Fähndrich, M. Barnett, and F. Logozzo. Embedded Contract Languages. In *SAC OOPS*, 2010.

C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. *SIGPLAN Not.*, 37(5):234–245, 2002.

C. Fournet, A. Gordon, and S. Maffeis. A Type Discipline for Authorization in Distributed Systems. In *IEEE Computer Security Foundations Symposium*, pages 31–48, 2007.

C. Fournet, N. Guts, and F. Zappa Nardelli. A formal implementation of value commitment. In *ESOP*, volume 4960 of *LNCS*, pages 383–397, 2008.

J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *VMCAI*, pages 363–379, 2005.

N. Guts, C. Fournet, and F. Zappa Nardelli. Reliable evidence: Auditability by typing. In *14th European Symposium on Research in Computer Security (ESORICS 2009)*, pages 168–183, 2009. To appear.

S. Haber and W. Stornetta. How to time-stamp a digital document. *Advances in Cryptology-CRYPT0'90*, pages 437–455, 1991.

A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. *ACM SIGOPS Operating Systems Review*, 41(6):188, 2007.

R. Hasan, R. Sion, and M. Winslett. Introducing secure provenance: problems and challenges. In *StorageSS*, 2007.

C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 1969.

ISO/IEC. Common criteria for information technology security evaluation. http://www.commoncriteriaportal.org/public/expert/index.php?menu=3, January 2004.

Recommendation X.509 (1997 E): Information Technology - Open Systems Interconnection - The Directory: Authentication Framework. ITU-T, June 1997.

R. Jagadeesan, A. Jeffrey, C. Pitcher, and J. Riely. Towards a Theory of Accountability and Audit. *ESORICS 2009*, 5789:152–167, 2009.

S. Jha, S. Katzenbeisser, C. Schallhart, H. Veith, and S. Chenney. Enforcing semantic integrity on untrusted clients in networked virtual environments. *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 179–186, 2007.

L. Jia, J. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. AURA: a programming language for authorization and audit. *ICFP*, pages 27–38, 2008.

S. Kremer and M. Ryan. Analysis of an electronic voting protocol in the applied pi calculus. In *ESOP*, pages 186–200, 2005.

S. Kremer, O. Markowitch, and J. Zhou. An intensive survey of fair non-repudiation protocols. *Computer Communications*, 25(17):1606–1621, 2002.

R. Küsters, T. Truderung, and A. Vogt. Accountability: Definition and relationship to verifiability. 2010.

P. Maniatis. *Historic integrity in distributed systems.* PhD thesis, Citeseer, 2003.

R. Milner. *Communicating and Mobile Systems: The Pi Calculus.* Cambridge University Press, 1999.

C. A. Neff. A verifiable secret shuffle and its application to e-voting. *ACM-CCS-2001*, 2001.

S. P. NIST. Generally accepted principles and practices for securing information technology systems. Available on http://csrc.nist.gov/publications/nistpubs/800-14/800-14.pdf, September 1996.

T. Okamoto and K. Ohta. Disposable zero-knowledge authentications and their applications to untraceable electronic cash. In *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 481–496. Springer, 1989.

J. Peha. Electronic commerce with verifiable audit trails. *INET*, 1999.

G. D. Plotkin. Denotational semantics with partial functions. Unpublished lecture notes, CSLI, Stanford University, July 1985.

Y. Régis-Gianas and F. Pottier. A Hoare logic for call-by-value functional programs. In *MPC*, pages 305–335, 2008.

M. Roe. Cryptography and evidence. *PhD thesis, University of Cambridge*, 1997.

P. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *PLDI*, pages 159–169, 2008.

B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, 2(2):159–176, May 1999.

A. Shamir, R.Rivest, and L. Adleman. Mental poker. *Mathematical Gardener*, pages 37–43, 1981.

N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in fine. In *ESOP*, pages 529–549, 2010.

Y. Tsiounis. Efficient electronic cash: New notions and techniques, 1997. Ph.D. thesis.

J. Vaughan, L. Jia, K. Mazurak, and S. Zdancewic. Evidence-based audit. In *IEEE Computer Security Foundations Symposium*, pages 177–191, 2008.

K. Vikram, A. Prateek, and B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 173–186. ACM, 2009.

B. R. Waters, D. Balfanz, G. Durfee, and D. K. Smetters. Building an encrypted and searchable audit log. In *Proceedings of Network and Distributed System Security Symposium 2004 (NDSS'04)*, San Diego, CA, February 2004.

D. N. Xu. Extended static checking for Haskell. In *Haskell*, pages 48–59, 2006.

W. Xu, D. Chadwick, and S. Otenko. A PKI Based Secure Audit Web Server. In *IASTED Communications, Network and Information and CNIS*, Phoenix, USA, November 2005. URL http://www.cs.kent.ac.uk/pubs/2005/2295.

A. Yumerefendi and J. Chase. Trust but verify: Accountability for network services. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 37. ACM, 2004.

A. Yumerefendi and J. Chase. The role of accountability in dependable distributed systems. In *Proceedings of HotDep*, 2005.

L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 236–250, Oakland, California, May 2003.

Appendix A

Preliminaries (complements)

Note We formalize our notations using the pretty-printing and the filtering functionalities of OTT (`http://moscova.inria.fr/~zappa/software/ott`) –a tool for writing definitions of programming languages.

A.1 Applied pi calculus

A.1.1 Reduction semantics

$\boxed{A \longrightarrow A'}$ A reduces to A'

$$\frac{}{c!\langle x\rangle.P_1 \mid c?(x).P_2 \longrightarrow P_1 \mid P_2} \quad \textsc{Comm}$$

$$\frac{}{\mathsf{if}\ M = M\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \longrightarrow P_1} \quad \textsc{If\_then}$$

$$\frac{}{\mathsf{if}\ M = M'\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \longrightarrow P_2} \quad \textsc{If\_else}$$

$$\frac{A_1 \longrightarrow A_1'}{A_1 \mid A_2 \longrightarrow A_1' \mid A_2} \quad \textsc{Ctx\_para\_l}$$

$$\frac{A \longrightarrow A'}{\nu\, r\,.\, A \longrightarrow \nu\, r\,.\, A'} \quad \textsc{Ctx\_new\_name}$$

$$\frac{A \longrightarrow A'}{\nu\, x\,.\, A \longrightarrow \nu\, x\,.\, A'} \quad \textsc{Ctx\_new\_var}$$

$$\frac{\begin{array}{l} A_1 \equiv A_1' \\ A_1' \longrightarrow A_2' \\ A_2' \equiv A_2 \end{array}}{A_1 \longrightarrow A_2} \quad \textsc{Redstruct}$$

A.1.2 Structural equivalence

The rules are given in Figure A.2.

A.1.3 Labelled semantics

Abadi and Fournet describe three flavours of labelled semantics:

(1) "reference" semantics where only channel names and variables of base types can be output directly, for other terms one has to create an active substitution and output the associated variable; it leads to an equivalence relation coinciding with observational equivalence (a barbed congruence);

(2) "naive" semantics where any term can be output directly; it leads to a finer, often unusable in applications, equivalence relation;

(3) "refined" semantics where only terms that can be derived by the environment can be output directly; it leads to the same equivalence relation as the "reference" semantics.

We recall that a variable x can be *derived* from the extended process A when, for some term M and extended process A', we have $A \equiv \{M/x\} \mid A'$. We use the third variant of the semantics where composite terms can be output but every exported restricted variable must be derivable by the environment.

Additionally, when receiving a term, we may need to export it as an active substitution. Since the term itself is already known by the environment, we deploy an active substitution without restricting its variable, provided that the variable is locally fresh. So we add the following labelled transition rule $A \xrightarrow{M/x} A|\{M/x\}$ when $x \notin fv(A)$ (rule Export).

The resulting labelled transition system is given in Figure A.1.

Figure A.1: Labelled transitions for applied pi calculus

$\boxed{A \xrightarrow{\psi} A'}$ A goes to A' with a label

$$\frac{A \longrightarrow A'}{A \xrightarrow{\tau} A'} \text{ Red}$$

$$\frac{}{c?(x).P \xrightarrow{c?(M)} P\{M/x\}} \text{ In}$$

$$\frac{}{c!\langle M\rangle.P \xrightarrow{c!<M>} P} \text{ Out}$$

$$\frac{A \xrightarrow{c!c'} A' \wedge c \neq c'}{\nu\, c'\,.\, A \xrightarrow{\nu\, r\,.\, c!<c'>} A'} \text{ OpenChannel}$$

$$\frac{A \xrightarrow{\nu\tilde{u}.\, c!<M>} A' \wedge r \in M \wedge r \notin u \quad r \text{ can be derived from } \nu\tilde{u}.\{M/z\} \mid A'}{\nu\, r\,.\, A \xrightarrow{\nu\, r\,.\, \nu\tilde{u}.\, c!<M>} A'} \text{ OpenVar}$$

$$\frac{A \xrightarrow{\psi} A' \wedge r \text{ does not occur in } \psi}{\nu\, r\,.\, A \xrightarrow{\psi} \nu\, r\,.\, A'} \text{ Scope}$$

$$\frac{x \text{ fresh for } A}{A \xrightarrow{M/x} A \mid \{M/x\}} \text{ Export}$$

$$\frac{A_1 \xrightarrow{\psi} A_1' \wedge BN(\psi) \cap FN(A_2) = \varnothing}{A_1 \mid A_2 \xrightarrow{\psi} A_1' \mid A_2} \text{ Par}$$

$$\frac{A_1 \equiv A_2 \wedge A_2 \xrightarrow{\psi} A_3 \wedge A_3 \equiv A_4}{A_1 \xrightarrow{\psi} A_4} \text{ Struct}$$

Figure A.2: Structural equivalence for applied pi calculus

$\boxed{A \equiv A'}$ A is structurally equivalent to A'

$$\frac{}{A \equiv A \mid l\,.\,\bot} \text{ Fresh\_loc}$$

$$\frac{}{a[P_1 \mid P_2] \equiv a[P_1] \mid a[P_2]} \text{ Par\_proc}$$

$$\frac{}{A \equiv A \mid p[0]} \text{ Par\_0}$$

$$\frac{}{A_1 \mid A_2 \mid A_3 \equiv A_1 \mid A_2 \mid A_3} \text{ Par\_A}$$

$$\frac{}{A \mid A' \equiv A' \mid A} \text{ Par\_C}$$

$$\frac{}{\nu\, u\,.\, p[0] \equiv p[0]} \text{ New\_0}$$

$$\frac{}{p[\nu\, c\,.\, P] \equiv \nu\, c\,.\, p[P]} \text{ New\_open}$$

$$\frac{}{\nu\, u\,.\, \nu\, u'\,.\, A \equiv \nu\, u'\,.\, \nu\, u\,.\, A} \text{ New\_name\_C}$$

$$\frac{u \notin FN(A) \cup FV(A)}{A \mid \nu\, u\,.\, A' \equiv \nu\, u\,.\, A \mid A'} \text{ New\_name\_Par}$$

$$\frac{}{\nu\, x\,.\, \{M/x\} \equiv 0} \text{ Alias}$$

$$\frac{}{\{M/x\} \mid A \equiv \{M/x\} \mid A\{M/x\}} \text{ Subst}$$

$$\frac{M = M'}{\{M/x\} \equiv \{M'/x\}} \text{ Rewrite}$$

A.2 RCF

The syntax of normalized RCF expressions and types is already given in Section 2.4 (Figures 2.2 and 2.3, respectively). Below, we include the evaluation, subtyping, and typing rules.

A.2.1 Evaluation

Heating: $A \Rrightarrow A'$

Axioms $A \equiv A'$ are read as both $A \Rrightarrow A'$ and $A' \Rrightarrow A$.

$A \Rrightarrow A$	(Heat Refl)
$A \Rrightarrow A''$ if $A \Rrightarrow A'$ and $A' \Rrightarrow A''$	(Heat Trans)
$A \Rrightarrow A' \Rightarrow \mathbf{let}\ x = A\ \mathbf{in}\ B \Rrightarrow \mathbf{let}\ x = A'\ \mathbf{in}\ B$	(Heat Let)
$A \Rrightarrow A' \Rightarrow (\nu a)A \Rrightarrow (\nu a)A'$	(Heat Res)
$A \Rrightarrow A' \Rightarrow (A \Rsh B) \Rrightarrow (A' \Rsh B)$	(Heat Fork 1)
$A \Rrightarrow A' \Rightarrow (B \Rsh A) \Rrightarrow (B \Rsh A')$	(Heat Fork 2)
$() \Rsh A \equiv A$	(Heat Fork ())
$a!M \Rrightarrow a!M \Rsh ()$	(Heat Msg ())
$\mathbf{assume}\ C \Rrightarrow \mathbf{assume}\ C \Rsh ()$	(Heat Assume ())
$a \notin fn(A') \Rightarrow A' \Rsh ((\nu a)A) \Rrightarrow (\nu a)(A' \Rsh A)$	(Heat Res Fork 1)
$a \notin fn(A') \Rightarrow ((\nu a)A) \Rsh A' \Rrightarrow (\nu a)(A \Rsh A')$	(Heat Res Fork 2)
$a \notin fn(B) \Rightarrow \mathbf{let}\ x = (\nu a)A\ \mathbf{in}\ B \Rrightarrow (\nu a)\mathbf{let}\ x = A\ \mathbf{in}\ B$	(Heat Res Let)
$(A \Rsh A') \Rsh A'' \equiv A \Rsh (A' \Rsh A'')$	(Heat Fork Assoc)
$(A \Rsh A') \Rsh A'' \Rrightarrow (A' \Rsh A) \Rsh A''$	(Heat Fork Comm)
$\mathbf{let}\ x = (A \Rsh A')\ \mathbf{in}\ B \equiv A \Rsh (\mathbf{let}\ x = A'\ \mathbf{in}\ B)$	(Heat Fork Let)

Reduction: $A \to A'$

$(\mathbf{rec}\ f : T.\ \mathbf{fun}\ x \to A)\ N \to A\{\mathbf{rec}\ f : T.\ \mathbf{fun}\ x \to A/f\}\{N/x\}$	(Red Rec Fun)
$(\mathbf{let}\ (x_1, x_2) = (N_1, N_2)\ \mathbf{in}\ A) \to A\{N_1/x_1\}\{N_2/x_2\}$	(Red Split)
$(\mathbf{match}\ M\ \mathbf{with}\ h\ x \to A\ \mathbf{else}\ B) \to \begin{cases} A\{N/x\} & \text{if } M = h\ N \text{ for some } N \\ B & \text{otherwise} \end{cases}$	(Red Match)
$M = N \to \begin{cases} \mathbf{true} & \text{if } M = N \\ \mathbf{false} & \text{otherwise} \end{cases}$	(Red Eq)
$a!M \Rsh a? \to M$	(Red Comm)
$\mathbf{assert}\ C \to ()$	(Red Assert)
$\mathbf{let}\ x = M\ \mathbf{in}\ A \to A\{M/x\}$	(Red Let Val)
$A \to A' \Rightarrow \mathbf{let}\ x = A\ \mathbf{in}\ B \to \mathbf{let}\ x = A'\ \mathbf{in}\ B$	(Red Let)
$A \to A' \Rightarrow (\nu a)A \to (\nu a)A'$	(Red Res)
$A \to A' \Rightarrow (A \Rsh B) \to (A' \Rsh B)$	(Red Fork 1)
$A \to A' \Rightarrow (B \Rsh A) \to (B \Rsh A')$	(Red Fork 2)
$A \to A'$ if $A \Rrightarrow B, B \to B', B' \Rrightarrow A'$	(Red Heat)

A.2.2 Subtyping

$recvar(E)$ denotes the type variables occurring in subtyping constraints of E.
$a \updownarrow T$ denotes an environment entry for a channel.
$\overline{A}$ denotes the formula extracted from the assumptions of A.

Subtype: $E \vdash P <: P'$, $E \vdash T <: T'$

(Sub Refl)
$$\frac{E \vdash P \quad recvar(E) \cap fnfv(P) = \varnothing}{E \vdash P <: P}$$

(Sub Fun)
$$\frac{E \vdash T_1' <: T_1 \quad E, x : T_1' \vdash T_2 <: T_2'[x/x']}{E \vdash (x : T_1 \to T_2) <: (x' : T_1' \to T_2')}$$

(Sub Pair)
$$\frac{E \vdash T_1 <: T_1' \quad E, x : T_1 \vdash T_2 <: T_2'[x/x']}{E \vdash (x : T \times_1 T_2) <: (x' : T_1' \times T_2')}$$

(Sub Refine)
$$\frac{E \vdash P <: P' \quad E, x : P, C \vdash C'[x/x']}{E \vdash (x : P)\{C\} <: (x' : P')\{C'\}}$$

(Sub Var)
$$\frac{E \vdash \diamond \quad (\alpha <: \alpha') \in E}{E \vdash \alpha <: \alpha'}$$

(Sub Rec)
$$\frac{E, \alpha <: \alpha' \vdash P <: P' \quad \alpha \notin fnfv(P') \quad \alpha' \notin fnfv(P)}{E \vdash (\mu\alpha.P) <: (\mu\alpha'.P')}$$

A.2.3 Typechecking

Typechecking: $E \vdash e : T$

(Typ Unit)
$$\frac{E \vdash \diamond}{E \vdash () : unit}$$

(Typ Var)
$$\frac{E \vdash \diamond \quad (x : T) \in E}{E \vdash x : T}$$

(Typ Refine)
$$\frac{E \vdash M : T \quad E \vdash C\{M/x\}}{E \vdash M : (x : T)\{C\}}$$

(Typ Pair)
$$\frac{E \vdash M_1 : T_1 \quad E, x : T_1 \vdash M_2 : T_2}{E \vdash (M_1, M_2) : x : T_1 \times T_2}$$

(Typ Assume)
$$\frac{E \vdash \diamond \quad fv(C) \subseteq dom(E)}{E \vdash \textbf{assume}\ C : (\_ : unit)\{C\}}$$

(Typ Assert)
$$\frac{E \vdash C}{E \vdash \textbf{assume}\ C : (\_ : unit)\{C\}}$$

(Typ Fun)
$$\frac{E \vdash x : T_1 \to T_2 <: T \quad E, f : T, x : T_1 \vdash e : T_2}{E \vdash \textbf{rec}\ f : T.(\textbf{fun}\ x \to e)\ :\ x : T_1 \to T_2}$$

(Typ App)
$$\frac{E \vdash M : x : T_1 \to T_2 \quad E \vdash N : T_1}{E \vdash (M\ N) : T_2\{N/x\}}$$

(Typ Let)
$$\frac{E \vdash e_1 : T_1 \quad E, x : T_1 \vdash e_2 : T \quad x \notin fv(T)}{E \vdash \textbf{let}\ x\ =\ e_1\ \textbf{in}\ e_2 : T}$$

(Typ Split)
$$\frac{E \vdash M_1 : x : T_1 \times T_2 \quad E, x_1 : T_1, x_2 : T_2, \_ : \{(x_1, x_2) = M_1\} \vdash M_2 : T \quad x_1, x_2 \notin fv(T)}{E \vdash \textbf{let}\ (x_1, x_2)\ =\ M_1\ \textbf{in}\ M_2 : T}$$

(Typ Cons)
$$\frac{h : (T, U) \quad E \vdash M : T \quad E \vdash U}{E \vdash h\ M : U}$$

(Typ Match Inl Inr Fold)
$$\frac{E \vdash M : T \quad h : (H, T) \quad E, x : H, \_ : \{h\ x = M\} \vdash A : U \quad E, \_ : \{\forall x.h\ x \neq M\} \vdash B : U}{E \vdash \textbf{match}\ M\ \textbf{with}\ h\ x \to A\ \textbf{else}\ B : U}$$

(Typ Res)
$$\frac{E, a \updownarrow T \vdash A : U \quad a \notin fn(U)}{E \vdash (\nu a)A : U}$$

(Typ Send)
$$\frac{E \vdash M : T \quad (a \updownarrow T) \in E}{E \vdash a!M : unit}$$

$$\frac{E \vdash \diamond \quad (a \updownarrow T) \in E}{E \vdash a? : T}\ \text{(Typ Recv)} \qquad \frac{E, \_ : \{\overline{A_2}\} \vdash A_1 : T_1 \quad E, \_ : \{\overline{A_1}\} \vdash A_2 : T_2}{E \vdash (A_1 \Rsh A_2) : T_2}\ \text{(Typ Fork)}$$

$$\frac{E \vdash e : T \quad E \vdash T <: T'}{E \vdash e : T'}\ \text{(Typ Subsum)} \qquad \frac{E \vdash e : T}{E \vdash (e : T) : T}\ \text{(Typ Annot)}$$

Appendix B

Value commitment

B.1 Semantics of the source language

B.1.1 Equational theory

$\boxed{M = M'}$ M is equal to M' in the equational theory

$$\frac{}{u = u}\ \textsc{Atom}$$

$$\frac{}{+_1((\ x\ +\ y\ )) = x}\ \textsc{Fst}$$

$$\frac{}{+_2((\ x\ +\ y\ )) = y}\ \textsc{Snd}$$

$$\frac{}{\mathsf{get\_idu}(x\,.\,\mathbf{Idc}\,(p\,)) = x\,.\,\mathbf{Idu}}\ \textsc{Idu\_of\_idc}$$

$$\frac{}{\mathsf{get\_idu}(x\,.\,\mathbf{Rd}\,(p\ v)) = x\,.\,\mathbf{Idu}}\ \textsc{Idu\_of\_rd}$$

$$\frac{}{\mathsf{get\_idc}(x\,.\,\mathbf{Rd}\,(p\ v)) = x\,.\,\mathbf{Idc}\,(p\,)}\ \textsc{Idc\_of\_rd}$$

$$\frac{}{\mathsf{get\_prin}(x\,.\,\mathbf{Idc}\,(p\,)) = p}\ \textsc{Prin\_of\_idc}$$

$$\frac{}{\mathsf{get\_prin}(x\,.\,\mathbf{Rd}\,(p\ v)) = p}\ \textsc{Prin\_of\_rd}$$

$$\frac{}{\mathsf{read}(x\,.\,\mathbf{Rd}\,(p\ v)) = v}\ \textsc{Read}$$

$$\frac{}{\mathsf{is\_idc}(x\,.\,\mathbf{Idc}\,(p\,)) = \mathsf{ok}}\ \textsc{Testidc}$$

$$\frac{}{\mathsf{is\_idu}(x\,.\,\mathbf{Idu}) = \mathsf{ok}}\ \textsc{Testidu}$$

$$\frac{}{\mathsf{is\_rd}(x\,.\,\mathbf{Rd}\,(p\ v)) = \mathsf{ok}}\ \textsc{Testrd}$$

B.1.2 Structural equivalence

$\boxed{A \equiv A'}$ A is structurally equivalent to A'

$$\frac{}{A \equiv A\ \mid\ l\,.\,\bot}\ \textsc{Fresh\_loc}$$

$$\frac{}{a[P_1 \mid P_2] \equiv a[P_1]\ \mid\ a[P_2]}\ \textsc{Par\_proc}$$

$$\frac{}{A \equiv A\ \mid\ p[0]}\ \textsc{Par\_0}$$

$$\frac{}{A_1\ \mid\ A_2\ \mid\ A_3 \equiv A_1\ \mid\ A_2\ \mid\ A_3}\ \textsc{Par\_A}$$

$$\frac{}{A \mid A' \equiv A' \mid A} \quad \textsc{Par\_C}$$

$$\frac{}{\nu\, u\,.\, p[0] \equiv p[0]} \quad \textsc{New\_0}$$

$$\frac{}{p[\nu\, c\,.\, P] \equiv \nu\, c\,.\, p[P]} \quad \textsc{New\_open}$$

$$\frac{}{\nu\, u\,.\, \nu\, u'\,.\, A \equiv \nu\, u'\,.\, \nu\, u\,.\, A} \quad \textsc{New\_name\_C}$$

$$\frac{u \notin FN(A) \cup FV(A)}{A \mid \nu\, u\,.\, A' \equiv \nu\, u\,.\, A \mid A'} \quad \textsc{New\_name\_Par}$$

$$\frac{}{\nu\, x\,.\, \{\, M \,/\, x \,\} \equiv 0} \quad \textsc{Alias}$$

$$\frac{}{\{\, M \,/\, x \,\} \mid A \equiv \{\, M \,/\, x \,\} \mid A\{^{M}/_{x}\}} \quad \textsc{Subst}$$

$$\frac{M = M'}{\{\, M \,/\, x \,\} \equiv \{\, M' \,/\, x \,\}} \quad \textsc{Rewrite}$$

B.1.3 Reduction semantics

$\boxed{A \longrightarrow A'}$ A reduces to A'

$$\frac{}{a[\mathsf{newloc}\,(x, y).P] \longrightarrow \nu\, l\,.\, (l.(a\,) \mid a[P\{^{l}/_{x}\}\{^{l\,.\,\mathbf{Idu}}/_{y}\}])} \quad \textsc{Newloc}$$

$$\frac{l \text{ fresh for } a[P]}{a[\mathsf{newloc}\,(x, y).P] \longrightarrow \nu\, l\,.\, (l\,.\,0\,(a\,) \mid a[P\{^{l}/_{x}\}\{^{l\,.\,\mathbf{Idu}}/_{y}\}])} \quad \textsc{Newloc\_ext}$$

$$\frac{Cap = 0 \vee Cap = \mathbf{Idu}}{l\,.\,Cap\,(a\,) \mid a[\mathsf{commit}\, M\, l\,(x).P] \longrightarrow l\,.\,Cap\,(a\ M) \mid a[P\{^{l\,.\,\mathbf{Rd}\,(a\ M)}/_{x}\}]} \quad \textsc{Commit\_ext}$$

$$\frac{}{l.(a\,) \mid a[\mathsf{commit}\, M\, l\,(x).P] \longrightarrow l.(a\, M) \mid a[P\{^{l\,.\,\mathbf{Rd}\,(a\ M)}/_{x}\}]} \quad \textsc{Commit}$$

$$\frac{}{a_1[c!\langle M\rangle.P_1] \mid a_2[c?(x).P_2] \longrightarrow a_1[P_1] \mid a_2[P_2\{^{M}/_{x}\}]} \quad \textsc{Comm}$$

$$\frac{\mathsf{adversary\ knows}\ H}{0 \longrightarrow \mathsf{resolving}(H)} \quad \textsc{AddRes}$$

$$\frac{}{\mathsf{resolving}(H) \longrightarrow 0} \quad \textsc{DelRes}$$

$$\frac{}{a[\mathsf{if}\ M\ =\ M\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2] \longrightarrow a[P_1]} \quad \textsc{If\_then}$$

$$\frac{M \neq M'}{a[\mathsf{if}\ M\ =\ M'\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2] \longrightarrow a[P_2]} \quad \textsc{If\_else}$$

$$\frac{A_1 \longrightarrow A_1'}{A_1 \mid A_2 \longrightarrow A_1' \mid A_2} \quad \textsc{Ctx\_par}$$

$$\frac{A \longrightarrow A'}{\nu\, u\,.\, A \longrightarrow \nu\, u\,.\, A'} \quad \textsc{Ctx\_new}$$

$$\frac{A_1 \equiv A_1' \wedge A_1' \longrightarrow A_2' \wedge A_2' \equiv A_2}{A_1 \longrightarrow A_2} \quad \textsc{Struct}$$

B.1.4 Ordering capabilities

$\boxed{C \preceq C'}$ C contains less data than C'

$$\frac{}{\bot \preceq 0\ ct} \quad 0$$

$$\frac{}{0\ ct \preceq \mathbf{Idu}\ ct}\ \text{0\_CT}$$

$$\frac{}{\mathbf{Idu}\ \mathsf{f}_u\ (ct) \preceq \mathbf{Idc}\ ct}\ \textsc{Idu}$$

$$\frac{}{\mathbf{Idc}\ \mathsf{f}_c\ (ct) \preceq \mathbf{Rd}\ ct}\ \textsc{Idc}$$

$$\frac{}{Cap\ (p\ H) \preceq Cap\ (p\ H\ M)}\ \textsc{Opt}$$

$$\frac{C_1 \preceq C_2 \quad C_2 \preceq C_3}{C_1 \preceq C_3}\ \textsc{Trans}$$

$$\frac{}{C \preceq C}\ \textsc{Refl}$$

$\boxed{M\ \rightsquigarrow_{cap} M'}$ state label M corresponds to the capability M'

$$\frac{}{u\,.\,\mathbf{Idu}\ (p\ H)\ \rightsquigarrow_{cap} u\,.\,\mathbf{Idu}\ ()}\ \textsc{Idu\_of\_lab}$$

$$\frac{}{u\,.\,\mathbf{Idc}\ (p\ H\ M)\ \rightsquigarrow_{cap} u\,.\,\mathbf{Idc}\ (p\ )}\ \textsc{Idc\_of\_lab}$$

$$\frac{}{u\,.\,\mathbf{Rd}\ (p\ \ M)\ \rightsquigarrow_{cap} u\,.\,\mathbf{Rd}\ (p\ \ M)}\ \textsc{Rd\_of\_lab}$$

$$\frac{}{Z\ \rightsquigarrow_{cap} Z}\ \textsc{Int\_of\_lab}$$

B.1.5 State transitions

$\boxed{C \xrightarrow{state\_lab} C'}$ label *state_lab* changes the state C into C'

$$\frac{}{C \xrightarrow{!\,C'} C \curlyvee C'}\ \textsc{Out}$$

$$\frac{C' \preceq C \wedge \mathsf{prin\_of}(C') \in \mathcal{A}}{C \xrightarrow{?C'} C}\ \textsc{In\_owned}$$

$$\frac{\mathsf{prin\_of}(C') \notin \mathcal{A}}{C \xrightarrow{?C'} C \curlyvee C'}\ \textsc{In}$$

B.1.6 Labelled semantics

$\boxed{A \xrightarrow{\alpha} A'}$ A reduces to A' with label α

$$\frac{}{a[c!\langle M\rangle.P] \xrightarrow{c\,!\,M} a[P]}\ \textsc{Send\_term}$$

$$\frac{}{a[c?(x).P] \xrightarrow{c?M} a[P\{^{M\sharp}/_x\}]}\ \textsc{Receive\_term}$$

$$\frac{A \xrightarrow{c?M} A' \wedge C_0 \xrightarrow{?\mathsf{locs}(l,M)} C_1}{l\,.\,C_0 \mid A \xrightarrow{c?M} l\,.\,C_1 \mid A'}\ \textsc{Up\_in}$$

$$\frac{A \xrightarrow{c\,!\,M} A' \wedge C_0 \xrightarrow{!\,\mathsf{locs}(l,M)} C_1}{l\,.\,C_0 \mid A \xrightarrow{c\,!\,M} l\,.\,C_1 \mid A'}\ \textsc{Up\_out}$$

$$\frac{A \xrightarrow{c\,!\,M} A' \wedge (\ c \neq r \wedge r \in M\ )}{\nu\, r\,.\,A \xrightarrow{\nu\, r\,.\ c\,!\,M} A'}\ \textsc{Open}$$

$$\frac{A_1 \longrightarrow A_2}{A_1 \xrightarrow{\tau} A_2}\ \textsc{Red}$$

$$\frac{A \xrightarrow{\alpha} A' \wedge c \notin \alpha}{\nu\, c\,.\, A \xrightarrow{\alpha} \nu\, c\,.\, A'} \textsc{Scope}$$

$$\frac{a[P_1] \xrightarrow{\alpha} a[P_1'] \wedge BN(\alpha) \cap FN(A_2) = \varnothing}{a[P_1] \mid A_2 \xrightarrow{\alpha} a[P_1'] \mid A_2} \textsc{Par}$$

$$\frac{A_1 \equiv A_2 \wedge A_2 \xrightarrow{\alpha} A_3 \wedge A_3 \equiv A_4}{A_1 \xrightarrow{\alpha} A_4} \textsc{Lab\_struct}$$

B.2 Proofs

This section is structured as follows. Subsection B.2.1 explains the dependencies between different parts of proofs and contains lemmas shared by the proofs of Theorems 3.1 and 3.2. Subsection B.2.2 proves functional adequacy. Subsection B.2.3 proves security.

In what follows we assume that the sets of bound and free names and variables that occur in labels and processes are disjoint. (The general case follows by renaming.)

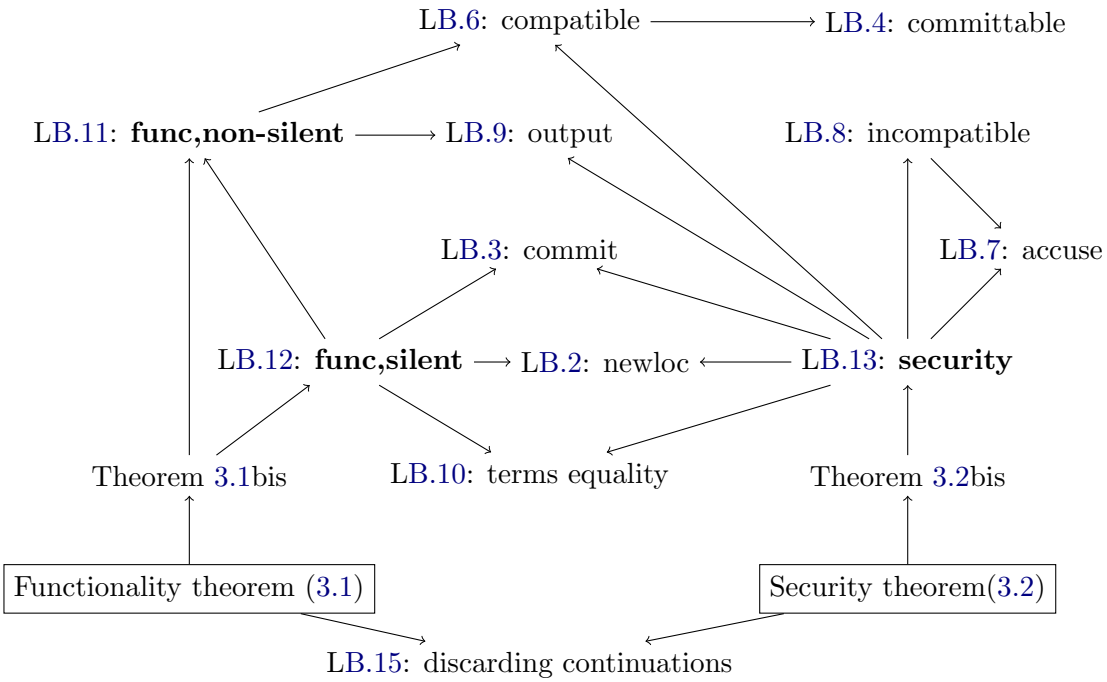
B.2.1 Preliminary lemmas

Important note on our main theorems In what follows we prove functional adequacy (Theorem 3.1bis) and security (Theorem 3.2bis) for the source systems extended with a resolution store. In Lemma B.15 we show that transitions due to this store are optional, and thus can be erased from any series of transitions between a usual system as defined in Section 3.2.4 and an extended system, to get transitions between two usual source systems. We then show that our proofs still hold for initial Theorem 3.1 and a slight variant of the initial Theorem 3.2.

In this subsection we anticipate several lemmas to factor proof cases of our theorems. We invite the reader to move to our theorems' proofs (B.2.2, B.2.3) and come back to these lemmas for references.

Lemma B.4 handles inputs of committable terms. Lemma B.6 handles inputs which do not allow to blame the environment: already known terms or fresh terms that successfully update the context. Lemmas B.8 handles input of a term, allowing to blame a principal. Lemma B.7 shows how a principal is accused. Lemma B.9 handles outputs. Lemma B.10 handles the translation of terms. Lemmas B.2 and B.3 handle the translation of newloc and commit constructs, respectively.

In the figure below arrows represent dependencies between lemmas.



Lemma B.1 (Discarding resolution store)**.** *For all well-formed source system A that contains no target continuations, let $\mathcal{T}$ be a non empty parallel composition of target continuations, we have that $A \mid \mathcal{T} \xrightarrow{\tau}{}^{*} A$*

Proof: Let A be a well-formed source system that contains no target continuations, let $\mathcal{T}$ be a set of target continuations. We proceed by strong induction on the number of target terms in $\mathcal{T}$.

(1) Let $\mathcal{T} = \mathsf{resolving}(M)$, then we can apply reduction rule DelRes, resulting in a silent transition $A \mid \mathsf{resolving}(M) \xrightarrow{\tau} A$.

(2) Let $\mathcal{T} = \mathsf{resolving}(M_1) \mid \ldots \mid \mathsf{resolving}(M_i) \mid \mathsf{resolving}(M)$. By the induction hypothesis, any of the terms $M_{1\ldots i}$ can be discarded: $A \mid \mathcal{T} \xrightarrow{\tau}{}^{*} A \mid \mathsf{resolving}(M)$. We can apply the induction again: $A \mid \mathsf{resolving}(M) \xrightarrow{\tau}{}^{*} A$, thus we have $A \mid \mathcal{T} \xrightarrow{\tau}{}^{*} A$.

□

Lemma B.2 (Creation of locations)**.** *The translation of* $\mathsf{newloc}\,(x, y).P$ *in an evaluation context run by principal a within a well-formed source system reduces as*

$$[\![\mathsf{newloc}\,(x, y).P]\!]_a \longrightarrow \nu\, s_l \,.\, \nu\, c_l \,.\, \nu\, l \,.\, ([\![a[P\{{}^{l}/_{x}\}\{{}^{l\,.\,\boldsymbol{Idu}}/_{y}\}]]\!] \mid [\![l \,.\, 0\,(a\,)]\!]) \tag{B.2.1}$$

Proof: By definition, we have

$$\begin{aligned}
&[\![\mathsf{newloc}\,(x, y).P]\!]_a \\
&= \nu\, s_{l'} \,.\, \nu\, c_l \,.\, \tau.(c_l!\langle\mathsf{None}\rangle \mid [\![P]\!]_a \{{}^{c_l}/_{c_x}\} \{{}^{s_{l'}}/_{s_x}\} \{{}^{\mathsf{h}(a+\mathsf{h}(s_{l'}))}/_{l}\} \{{}^{\mathsf{idu}(l)}/_{y}\}) \\
&\equiv \nu\, l \,.\, \nu\, s_l \,.\, \nu\, s_{l'} \,.\, \nu\, c_l \,.\, \tau.(c_l!\langle\mathsf{None}\rangle \\
&\mid [\![P]\!]_a \{{}^{c_l}/_{c_x}\} \{{}^{s_{l'}}/_{s_l}\} \{{}^{s_l}/_{s_x}\} \{{}^{\mathsf{h}(a+\mathsf{h}(s_l))}/_{l}\} \{{}^{\mathsf{idu}(l)}/_{y}\}) && \text{(B.2.2)} \\
&\equiv \nu\, s_l \,.\, \nu\, c_l \,.\, \nu\, l \,.\, \tau.((c_l!\langle\mathsf{None}\rangle \mid (\nu\, s_{l'} \,.\, \{s_{l'}/s_l\}) \mid \{\mathsf{h}(a + \mathsf{h}(s_l))/l\}) \\
&\mid [\![P]\!]_a \{{}^{c_l}/_{c_x}\} \{{}^{s_l}/_{s_x}\} \{{}^{\mathsf{idu}(l)}/_{y}\}) && \text{(B.2.3)} \\
&\equiv \nu\, s_l \,.\, \nu\, c_l \,.\, \nu\, l \,.\, \tau.([\![l \,.\, 0\,(a\,)]\!] \mid [\![P\{{}^{l}/_{x}\}\{{}^{l\,.\,\mathbf{Idu}}/_{y}\}]\!]_a) && \text{(B.2.4)} \\
&\longrightarrow \nu\, s_l \,.\, \nu\, c_l \,.\, \nu\, l \,.\, ([\![l \,.\, 0\,(a\,)]\!] \mid [\![P\{{}^{l}/_{x}\}\{{}^{l\,.\,\mathbf{Idu}}/_{y}\}]\!]_a) && \text{(B.2.5)}
\end{aligned}$$

We introduce a substitution $\{s_{l'}/s_l\}$ for a fresh name s_l; we restrict s_l and l on toplevel in B.2.2, since they are fresh. We introduce active substitutions for l and s_l; as s'_l does not occur in P, we restrict it locally (B.2.3). In (B.2.4), we fold the translation of an uncommitted location and of the continuation process that has received a reference to that location.

At the target level, the channel name c_l and the cell identifier s_l play the same role as the location name l at the source level. They are used within the commit primitive and its translation, and nowhere else.

Finally, (B.2.5) is the reduction of τ construct. □

Lemma B.3 (Commitment of a location)**.** *The translation of* commit $V\ x\ (x').P$ *in an evaluation context run by principal a within a well-formed source system is*

$$[\![\mathsf{commit}\ V\ x\ (x').P]\!]_a =$$
$$c_x?(\_).\nu\, v_x\,.\,\nu\, w_x\,.\,(\varsigma(\mathsf{h}(s_x), \mathsf{h}(s_x + [\![V]\!]))_a \mid [\![P]\!]_a\,\{^{\mathsf{rd}(a\,,\,s_x\,,\,[\![V]\!]\,,\,w_x)}/_{x'}\})$$

Proof: By structural equivalence, using the definition of ς. □

Definition B.2.1 (Context compatibility)**.** A well-formed source context $\mathcal{C}_a$ of the form

$$\mathcal{C}_a[\_] = \nu\mathcal{N}.\left(\textstyle\prod_{l\in\mathcal{L}} l.C_l \mid \prod_{a'\in\mathcal{A}\setminus\{a\}} a'[P_{a'}] \mid a[\_] \mid \phi \mid \mathcal{T}\right)$$

is *compatible* with label $c?M$ if for all $l \in \mathcal{L}$, there is C'_l such that $C_l \xrightarrow{?\mathsf{locs}(l,M)} C'_l$.

Definition B.2.2 (Context update)**.** For any well-formed source context $\mathcal{C}_a$ of the form

$$\mathcal{C}_a[\_] = \nu\mathcal{N}.\left(\textstyle\prod_{l\in\mathcal{L}} l.C_l \mid \prod_{a'\in\mathcal{A}\setminus\{a\}} a'[P_{a'}] \mid a[\_] \mid \phi \mid \mathcal{T}\right)$$

for any label $c?M$, $\mathcal{C}_a \oplus M$ denotes the well-formed context obtained by updating its locations according to M:

$$(\mathcal{C}_a \oplus M)[\_] \;=\; \nu\mathcal{N}.\;\left(\textstyle\prod_{l\in\mathcal{L}} l.(C_l + locs(l,M)) \mid \prod_{a'\in\mathcal{A}\setminus\{a\}} a'[P_{a'}] \mid a[\_] \mid \phi \mid \mathcal{T}\right)$$

Note All committable terms are compatible with any well formed source context (exception $l.Idu$ when $C_l = 0\ ct$).

Lemma B.4 (Parsing a compatible committable term)**.** *For any well-formed source context $\mathcal{C}_a$ for any source label M such that term $M^\sharp$ is committable, for a target process P,*

$$[\![\mathcal{C}_a]\!][\mathsf{parse_c}\ [\![M^\sharp]\!]\ P] \xrightarrow{\tau}{}^{*}\ (\mathcal{C}_a \oplus M)[P]$$

Proof: We proceed by induction on the structure of label M. We recall the definition of $\mathsf{parse_c}$:

$$\begin{array}{l}\mathsf{parse_c}\ x\ P = \\ \quad \mathsf{if\ is\_idu}(x) \;=\; \mathsf{ok\ then}\ P \\ \quad\quad \mathsf{else\ if\ is\_prin}(x) \;=\; \mathsf{ok\ and\ is\_pk}(x) \;=\; \mathsf{ok\ then}\ P \\ \quad\quad\quad \mathsf{else\ if\ is\_pair}(x) \;=\; \mathsf{ok\ then\ parse_c}\ (+_1\ x)\ (\mathsf{parse_c}\ (+_2\ x)\ P) \\ \quad\quad\quad\quad \mathsf{else}\ r!\langle\mathsf{None}\rangle\end{array}$$

(1) If $M = l\,.\,\mathbf{Idu}$ then $\mathsf{prin\_of}(\mathbf{Idu}) \in \mathcal{A}$ and $\mathsf{locs}(l, M) = M^\sharp = \mathbf{Idu}$. Only rule In_owned applies to the assumption $C_l \xrightarrow{?\mathbf{Idu}} C'_l$, so necessarily$\mathbf{Idu} \preceq C_l$ and $C'_l = C_l$. The translation of C_l must contain an active substitution $\{H/l\}$, so we have $[\![M^\sharp]\!] = \mathsf{idu}(l)$. The first test (is_idu) in parse succeeds, so we have $[\![\mathcal{C}_a]\!][\mathsf{parse_c\ idu}(l)\ P] \xrightarrow{\tau} [\![\mathcal{C}_a]\!][P]$.

(2) If $M = l\,.\,\mathbf{Idu}\,(e\,H)$ then $e = \mathsf{prin\_of}(\mathbf{Idu}) \notin \mathcal{A}$ and $\mathsf{locs}(l, M) = M^\sharp = \mathbf{Idu}\,(e\,H)$. Only rule In applies to the assumption $C_l \xrightarrow{?\mathbf{Idu}\,(e\,H)} C'_l$, so $C'_l = C_l \curlyvee \mathbf{Idu}\,(e\,H)$. (For all other locations $l_0 \in \mathcal{L} \setminus \{l\}$, $\mathsf{locs}(l_0, M) = \bot$, so $C'_{l_0} = C_{l_0}$.

For the update of l two cases are possible.

- If $\mathbf{Idu}\,(e\,H) \preceq C_l$, then $C'_l = C_l$ and the translation of $l.C_l$ already contains an active substitution $\{H/l\}$. Like in the previous case, we have $[\![M^\sharp]\!] = \mathsf{idu}(l)$ and $[\![\mathcal{C}_a]\!][\mathsf{parse_c}\,\mathsf{idu}(l)\,P] \xrightarrow{\tau} [\![\mathcal{C}_a]\!][P]$.

- Otherwise, the received location l is unknown (and fresh for the source system and its translation). We have $[\![M^\sharp]\!] = \mathsf{idu}(H)$ and $[\![\mathcal{C}_a]\!][\mathsf{parse_c}\,\mathsf{idu}(H)\,P] \xrightarrow{\tau} [\![\mathcal{C}_a]\!][P]$. By rule Fresh_loc, $\mathcal{C}_a[\_] \equiv l\,.\,\bot \mid \mathcal{C}_a[\_]$.

In the translation we introduce an active substitution for l, then fold the updated location $C'_l = \bot \curlyvee \mathbf{Idu}\,(e\,H) = \mathbf{Idu}\,(e\,H)$.

$$[\![l\,.\,\bot]\!] \mid P \equiv (\nu\, l\,.\,\{H/l\}) \mid P \equiv \nu\, l\,.\,[\![l\,.\,\mathbf{Idu}\,(e\,H)]\!]|P$$

We thus have $[\![\mathcal{C}_a]\!][P] \equiv [\![\mathcal{C}'_a]\!][P]$.

(3) If M is a name of principal p, we have $M^\sharp = M$, for all $l \in \mathcal{L}$, and $\mathsf{locs}(l, M) = \bot$. The translation of the context contains an active substitution $\{\mathsf{prin}(\mathsf{pk}(x_p))/p\}$ where x_p is a nonce for honest principals p, or an arbitrary term for external principals. Thus in $\mathsf{parse_c}$, the first $\mathsf{is\_idu}$ test fails, the next two ($\mathsf{is\_prin}$ and $\mathsf{is\_pk}$) succeed, and we have: $[\![\mathcal{C}_a]\!][\mathsf{parse_c}\,[\![M^\sharp]\!]\,P] \xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} [\![\mathcal{C}_a]\!][P]$.

(4) If M is a pair $M_1 + M_2$, then $[\![M]\!] = [\![M_1]\!] + [\![M_2]\!]$. In $\mathsf{parse_c}$, the first two tests fail, the third one ($\mathsf{is\_pair}$) succeeds, and we have:

$$[\![\mathcal{C}_a]\!][\mathsf{parse_c}\,[\![M^\sharp]\!]\,P] \xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} [\![\mathcal{C}_a]\!][\mathsf{parse_c}\,[\![M_1]\!]\,(\mathsf{parse_c}\,[\![M_2]\!]\,P)].$$

By assumption, for all $l \in \mathcal{L}$, there is $C'_l = C_l \curlyvee \mathsf{locs}(l, M)$. Since $\mathsf{locs}(l, M) = \mathsf{locs}(l, M_1) \curlyvee \mathsf{locs}(l, M_2))$, we know that for all $l \in \mathcal{L}$, $C_l \curlyvee \mathsf{locs}(l, M_1)$ exists. We first apply the induction hypothesis on M_1; context $\mathcal{C}''_a$ is well-formed, and for all $l \in \mathcal{L}'$ there is $C''_l = C_l \curlyvee \mathsf{locs}(l, M_1)$, and so $C''_l \curlyvee \mathsf{locs}(l, M_2) = C'_l$. We can now apply the induction on M_2 and obtain another well-formed context $\mathcal{C}'_a$:

$$[\![\mathcal{C}_a]\!][\mathsf{parse_c}\,[\![M_1]\!]\,(\mathsf{parse_c}\,[\![M_2]\!]\,P)] \xrightarrow{\tau}{}^* [\![\mathcal{C}''_a]\!][\mathsf{parse_c}\,[\![M_2]\!]\,P] \xrightarrow{\tau}{}^* [\![\mathcal{C}'_a]\!][P].$$

□

Lemma B.5 (Decomposing parse for pairs)**.** *For any well-formed source contexts $\mathcal{C}_a$, $\mathcal{C}'_a$, for any marshallable term M,for a target process P,*

if $[\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M]\!]\,P] \longrightarrow_D^* [\![\mathcal{C}'_a]\!][P]$ *then*

(1) $[\![\mathcal{C}_a]\!][\mathsf{parse}_1\,[\![M]\!]\,P] \longrightarrow_D^* [\![\mathcal{C}_a]\!][P]$

(2) $[\![\mathcal{C}_a]\!][\mathsf{parse}_2\,[\![M]\!]\,] \longrightarrow_D^* [\![\mathcal{C}'_a]\!][0]$

Proof: Let $\mathcal{C}_a, \mathcal{C}'_a$ be well-formed source contexts M a marshallable term M,P a target process, and $[\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M]\!]\,P] \longrightarrow_D^* [\![\mathcal{C}'_a]\!][P]$. By definition,

$$[\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M]\!]\,P] = [\![\mathcal{C}_a]\!][\mathsf{parse}_1\,[\![M]\!]\,(P \mid \mathsf{parse}_2\,[\![M]\!]\,)] \longrightarrow_D^* [\![\mathcal{C}_a]\!][(P \mid \mathsf{parse}_2\,[\![M]\!]\,)]$$

(1) The process $\mathsf{parse}_2\,[\![M]\!]$ is passive so we also have, $[\![\mathcal{C}_a]\!][\mathsf{parse}_1\,[\![M]\!]\,P] \longrightarrow_D^* [\![\mathcal{C}_a]\!][P]$.

(2) $[\![\mathcal{C}_a]\!][P \mid \mathsf{parse}_2\,[\![M]\!]\,] \longrightarrow_D^* [\![\mathcal{C}'_a]\!][P]$. The process P is passive, so we also have

$$[\![\mathcal{C}_a]\!][\mathsf{parse}_2\,[\![M]\!]\,] \longrightarrow_D^* [\![\mathcal{C}'_a]\!][0].$$

□

The next lemma states that the parse function succeeds, and all locations are updated according to the source rules, if these rules apply to the received term.

Lemma B.6 (Parsing a compatible term from environment)**.** *For any well-formed source context $\mathcal{C}_a$ for any source label M , for a target process P, if $\mathcal{C}_a$ is compatible with M then*

$$[\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M^\sharp]\!]\,P] \xrightarrow{\tau}{}^{*}\ (\mathcal{C}_a \oplus M)[P].$$

Proof: We proceed by induction on the structure of label M. We recall the definition of parse:

$$\begin{array}{l}
\mathsf{parse}_1\ x\ P = \\
\quad \mathsf{if}\ \mathsf{is_rd}(x)\ =\ \mathsf{ok}\ \mathsf{then} \\
\quad\quad \mathsf{if}\ \mathsf{check_idc}(\mathsf{get_idc}(x))\ \mathsf{then}\ \mathsf{parse}_{\mathsf{c}}\ \mathsf{read}(x)\ P\ \mathsf{else}\ r!\langle\mathsf{None}\rangle \\
\quad\quad \mathsf{else\ if}\ \mathsf{is_idc}(x)\ =\ \mathsf{ok}\ \mathsf{then}\ \mathsf{if}\ \mathsf{check_idc}(x)\ \mathsf{then}\ P\ \mathsf{else}\ r!\langle\mathsf{None}\rangle \\
\quad\quad\quad \mathsf{else\ if}\ \mathsf{is_pair}(x)\ =\ \mathsf{ok}\ \mathsf{then}\ \mathsf{parse}_1\ (+_1\ x)\ (\mathsf{parse}_1\ (+_2\ x)\ P) \\
\quad\quad\quad\quad \mathsf{else}\ \mathsf{parse}_{\mathsf{c}}\ x\ P \\
\mathsf{parse}_2\ x\ = \\
\quad \mathsf{if}\ \mathsf{is_rd}(x)\ =\ \mathsf{ok}\ \mathsf{then}\ \mathsf{repl}\ \mathit{log}!\langle\mathsf{get_idc}(x)\rangle \\
\quad \mathsf{else\ if}\ \mathsf{is_idc}(x)\ =\ \mathsf{ok}\ \mathsf{then}\ \mathsf{repl}\ \mathit{log}!\langle x\rangle \\
\quad\quad \mathsf{else\ if}\ \mathsf{is_pair}(x)\ =\ \mathsf{ok}\ \mathsf{then}\ (\mathsf{parse}_2\ (+_1\ x)\ \mid \mathsf{parse}_2\ (+_2\ x)\ ) \\
\mathsf{parse}\ x\ P = \mathsf{parse}_1\ x\ (P \mid \mathsf{parse}_2\ x\ )
\end{array}$$

(1) If $M = l.\mathbf{Rd}\,(a\ V)$ then $a \in \mathcal{A}$, $\mathsf{locs}(l, M) = \mathbf{Rd}\,(a\ V) \curlyvee \mathsf{locs}(l, M)$, and $M^\sharp = M$. Only rule In_owned applies to the assumption $C_l \xrightarrow{?\mathbf{Rd}\,(a\ V)} C'_l$, so necessarily$\mathbf{Rd}\,(a\ V) \preceq C_l$ and $C'_l = C_l = \mathbf{Rd}\,(a\ V)$. The translation of C_l must assign (with active substitutions) $s_l, \{v_l = \mathsf{h}(s_l) + \mathsf{h}(s_l + [\![V]\!])\}, \{w_l = \mathsf{sign}(v_l\,, \mathsf{sk}(m_p))\}$ and logs $\mathsf{idc}(p\,, v_l\,, w_l)$ such that $[\![M^\sharp]\!] = \mathsf{rd}(p\,, s_l\,, [\![V]\!]\,, w_l)$.
In the call $\mathsf{parse}\,\mathsf{rd}(p\,, s_l\,, [\![V]\!]\,, w_l)\,P$, the first test (is_rd) succeeds yielding a silent transition. The second test is evaluated by unfolding the definition of check_idc and applying the equations that define get_idc,idc$_1$,idc$_2$, idc$_3$,check:

$$\begin{array}{c}
\mathsf{check_idc}(\mathsf{get_idc}(\mathsf{rd}(p\,, s_l\,, [\![V]\!]\,, w_l))) \\
= \mathsf{check_idc}(\mathsf{idc}(p\,, \mathsf{h}(s_l) + \mathsf{h}(s_l + [\![V]\!])\,, w_l)) \\
= \mathsf{verify}(\mathsf{h}(s_l) + \mathsf{h}(s_l + [\![V]\!])\,, w_l\,, p)\ =\ \mathsf{ok}.
\end{array}$$

The checks succeed, so $[\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M^\sharp]\!]\,P] \xrightarrow{\tau}\xrightarrow{\tau} P_2$ with the continuation

$$P_2 = [\![\mathcal{C}_a]\!][\mathsf{parse}_{\mathsf{c}}\,\mathsf{read}([\![M^\sharp]\!])\,(P \mid \mathsf{parse}_2\,[\![M^\sharp]\!]\,)].$$

Since $\mathsf{read}([\![M^\sharp]\!]) = [\![V]\!]$, V is committable and $\mathsf{locs}(l, V) \preceq C_l$, we can apply Lemma B.4:

$$P_2 \xrightarrow{\tau}{}^{*}\ [\![\mathcal{C}_a]\!][P \mid \mathsf{parse}_2\,[\![M^\sharp]\!]\,]$$

The first test succeeds in parse$_2$ succeeds, and reduces to replicated output on log, which we discard since it is already contained in $[\![l\,.\,\mathcal{C}]\!]$:

$$[\![\mathcal{C}_a]\!][P \mid \mathsf{parse}_2\,[\![M^\sharp]\!]\,] \xrightarrow{\tau} [\![\mathcal{C}_a]\!][P \mid \mathsf{repl}\ \mathit{log}!\langle\mathsf{idc}(p\,, v_l\,, w_l)\rangle] \equiv [\![\mathcal{C}_a]\!][P]$$

We use the equivalence Repl_repl:

$$\mathsf{repl}\ \mathit{log}!\langle\mathsf{idc}(p\,, v_l\,, w_l)\rangle \mid \mathsf{repl}\ \mathit{log}!\langle\mathsf{idc}(p\,, v_l\,, w_l)\rangle \equiv \mathsf{repl}\ \mathit{log}!\langle\mathsf{get\_idc}([\![M]\!])\rangle$$

(2) If $M = l\,.\,\mathbf{Rd}\,(e_0\ H\ V)$, then $e0 \notin \mathcal{A}$, $\mathsf{locs}(l, M) = \mathbf{Rd}\,(e_0\ H\ V)\curlyvee\mathsf{locs}(l, M)$, and $M^\sharp = l.\mathbf{Rd}\,(e_0\ V)$. Only rule In applies to the assumption $C_l \xrightarrow{?\mathbf{Rd}\,(e_0\ H\ V)} C'_l$, so necessarily $C'_l = C_l\curlyvee\mathbf{Rd}\,(e_0\ H\ V)$.

Two cases depending on C_l are possible.

- If $\mathbf{Rd}\,(e_0\ H\ V) \preceq C_l$ then the translation of C_l must contain active substitutions s_l, v_l, w_l such that $[\![M^\sharp]\!] = \mathsf{rd}(p\,,\ s_l\,,\ [\![V]\!]\,,\ w_l)$. Then no update is done and *parse* reduces exactly as in case $M = \mathbf{Rd}\,(a\ V)$.

- Otherwise $C_l \preceq \mathbf{Rd}\,(e_0\ H\ V)$ and $C_l \neq \mathbf{Rd}\,(e_0\ H\ V)$, the environment is sending new data.

$$[\![M]\!] = \mathsf{rd}(e_0\,,\ H\,,\ [\![V]\!]\,,\ \mathsf{sign}(\mathsf{h}(H) + \mathsf{h}(H + [\![V]\!])\,,\ \mathsf{sk}(m_{e_0}))).$$

The first test in parse_1 and the check_idc test succeed, so the process reduces to P' in the same context. By assumption, context $\mathcal{C}_a$ is compatible with M, so there exists $C''_l = C_l\curlyvee\mathsf{locs}(l, M)$, and by definition $\mathsf{locs}(l, M) = \mathbf{Rd}\,(e_0\ H\ V)\curlyvee\mathsf{locs}(l, V)$, so $C_l\curlyvee\mathsf{locs}(l, V)$ exists. Since $\mathsf{read}([\![M^\sharp]\!]) = [\![V]\!]$ and V is committable we can apply Lemma B.4

$$[\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M^\sharp]\!]\,P] \xrightarrow{\tau}{}^{*}\ [\![\mathcal{C}_a]\!][\mathsf{parse_c}\,[\![V]\!]\,(P \mid \mathsf{parse}_2\,[\![M^\sharp]\!]\,)] \xrightarrow{\tau}{}^{*}\ (\mathcal{C}_a \oplus V)[P \mid \mathsf{parse}_2\,[\![M^\sharp]\!]\,]$$

By We have that $(\mathcal{C}_a \oplus M)$ is compatible with V. The first test in parse_2 succeeds, and reduces to replicated output on *log*:

$$\begin{aligned}
&P \mid \mathsf{parse}_2\,[\![M^\sharp]\!] \xrightarrow{\tau} P \mid \mathsf{repl}\ log!\langle\mathsf{get_idc}([\![M^\sharp]\!])\rangle\\
&= \nu\, v'_l\,.\,\nu\, w'_l\,.\,(P \mid \mathsf{repl}\ log!\langle\mathsf{idc}(e_0\,,\ v'_l\,,\ w'_l)\rangle\,\sigma) = P'\\
&\text{with } \sigma = \{^{\mathsf{h}(H)+\mathsf{h}(H+[\![V]\!])}/_{v'_l}\}\{^{\mathsf{sign}(v'_l\,,\mathsf{sk}(m_{e_0}))}/_{w'_l}\}
\end{aligned} \tag{B.2.6}$$

We anticipate some fresh local substitutions (B.2.6). We compute the remaining update for location l, by applying structural equivalences to P' and the concerned part of the context $[\![l\,.\,C'_l]\!] \mid P'$.

Here are different cases depending on C'_l.

– If $C'_l = \mathbf{Idc}\,(e_0\,(\,M_1\ +\ M_2\,)\ V)$ satisfying the definition of f_c function, that is $\mathbf{h}(H)+\mathbf{h}(H+[\![V]\!]) = M_1+M_2$, we have

$$\begin{aligned}
&[\![l\,.\,C'_l]\!] \mid P'\\
&= [\![l\,.\,\mathbf{Idc}\,(e_0\,(\,M_1\ +\ M_2\,)\ V)]\!] \mid P'\\
&= \varphi(M_1, M_2)_{e_0} \mid (P \mid \mathsf{repl}\ log!\langle\mathsf{idc}(e_0\,,\ v_l\,,\ w_l)\rangle\,\{^{\mathsf{h}(H)+\mathsf{h}(H+[\![V]\!])}/_{v_l}\}\{^{\mathsf{sign}(v_l\,,\mathsf{sk}(m_{e_0}))}/_{w_l}\}) && \text{(B.2.7)}\\
&\equiv \varphi(M_1, M_2)_{e_0} \mid P && \text{(B.2.8)}\\
&\equiv \nu\, s_l\,.\,((\varphi(M_1, M_2)_{e_0} \mid \{H/s_l\}) \mid P) && \text{(B.2.9)}\\
&\equiv [\![l\,.\,\mathbf{Rd}\,(e_0\ H\ V)]\!] \mid P && \text{(B.2.10)}\\
&\equiv [\![l\,.\,C''_l]\!] \mid P
\end{aligned}$$

In (B.2.7) we translate **Idc**, and as we unfold P' we equate local substitutions v'_l, w'_l to the active substitutions v_l, w_l defined by φ; in (B.2.8) we use Repl_repl to merge the replication with that contained within φ; in (B.2.9) we introduce fresh substitution s_l for H and restrict it on top level; in (B.2.10) we fold the translation of **Rd** tagged location. We have indeed $l.C''_l = l\,.\,\mathbf{Rd}\,(e_0\ H\ V) = l.\mathbf{Idc}\,(e_0\,(\,M_1\ +\ M_2\,)\ V)\curlyvee\mathbf{Rd}\,(e_0\ H\ V) = l.C'_l\curlyvee\mathsf{locs}(l, M)$. We thus constructed a well-formed context $\mathcal{C}''_a$ such that $\forall l_0 \in \mathcal{L} \setminus \{l\}$ $C''_{l_0} = C_l\curlyvee\mathsf{locs}(l_0, M)$.

– If $C'_l = \mathbf{Idu}\,(e_0\ H')$ satisfying the definition of f_c and f_u functions, that is $\mathbf{h}(e_0{+}\mathbf{h}(H)) = H'$, we have

$$\begin{aligned}
&[\![l\,.\,\mathbf{Idu}\,(e_0\ H')]\!] \mid P' \\
&\equiv \{H'/l\} \mid (\varsigma(\mathsf{h}(H), \mathsf{h}(H + [\![V]\!]))_{e_0} \mid P) && \text{(B.2.11)}\\
&\equiv \{H/s_l\} \mid \varphi(\mathsf{h}(H), \mathsf{h}(H + [\![V]\!]))_{e_0} \mid P && \text{(B.2.12)}\\
&\equiv [\![l\,.\,\mathbf{Rd}\,(e_0\ H\ V)]\!] | P && \text{(B.2.13)}\\
&\equiv [\![l\,.\,C''_l]\!] \mid P
\end{aligned}$$

In (B.2.11) we introduce fresh names s_l, v_l, w_l and transform substitutions from σ into (not restricted) active substitutions which fold the ς process. In (B.2.12) we introduce a fresh (not restricted) active substitution for s_l, and fold the definition of φ process; in (B.2.13) we fold the translation of **Rd** tagged location. We have indeed $l.C''_l = l\,.\,\mathbf{Rd}\,(e_0\ H\ V) = l.\mathbf{Idu}\,(e_0\ H')\,\Upsilon\,\mathbf{Rd}\,(e_0\ H\ V) = l.C'_l\,\Upsilon\,\mathsf{locs}(l, M)$.

– If $C'_l = \bot$ then we have

$$\begin{aligned}
&[\![l\,.\,\bot]\!] \mid P' \\
&\equiv 0 \mid (\{H/s_l\} \mid \varphi(\mathsf{h}(H), \mathsf{h}(H + [\![V]\!]))_{e_0} \mid P\,\{^{\mathsf{rd}(e_0\,,\,s_l\,,\,[\![V]\!]\,,\,w_l)}/_x\})\\
&\equiv ([\![l\,.\,\mathbf{Rd}\,(e_0\ H\ V)]\!] | P\{^{[\![l\,.\,\mathbf{Rd}\,(e_0\ V)]\!]}/_x\})\\
&\equiv [\![l\,.\,C''_l]\!] \mid P\,\{^{[\![M^\sharp]\!]}/_x\}
\end{aligned}$$

We introduce (not restricted) fresh variables l, s_l, v_l, w_l to represent location state of l. The rule Up_in applies: $C''_l = \mathbf{Rd}\,(e_0\ H\ V) = \bot\,\Upsilon\,\mathbf{Rd}\,(e_0\ H\ V) = C'_l\,\Upsilon\,C$.

In each case we thus have $[\![\mathcal{C}'_a]\!][P'] \equiv [\![\mathcal{C}''_a]\!][P\,\{^{[\![M]\!]}/_x\}]$. So

$$[\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M]\!]\ P\,\{^{[\![M]\!]}/_x\}] \xrightarrow{\tau}{}^{*}\ [\![\mathcal{C}''_a]\!][P\,\{^{[\![M]\!]}/_x\}]$$

where all locations occurring in M are updated correctly.

(3) If $M = l.\mathbf{Idc}\,(a\,)$ then $a \in \mathcal{A}$, $\mathsf{locs}(l, M) = M^\sharp = \mathbf{Idc}\,(a\,)$. Only rule In_owned applies to the assumption $C_l \xrightarrow{?\mathbf{Idc}\,(a\,)} C'_l$, so necessarily$\mathbf{Idc}\,(a\,) \preceq C_l$ and $C'_l = C_l$. Thus we have $[\![M]\!] = \mathsf{idc}(a\,,\,v_l\,,\,w_l)$ such that the translation of $l.C_l$ assigns (with active substitutions) $v_l = \mathsf{h}(s_l) + \mathsf{h}(s_l + [\![V]\!])$ and $w_l = \mathsf{sign}(v_l\,,\,\mathsf{sk}(m_a))$ and also logs $\mathsf{idc}(a\,,\,v_l\,,\,w_l)$.

In $\mathsf{parse}_1\ \mathsf{idc}(a\,,\,v_l\,,\,w_l)\ P\,\{^{[\![M^\sharp]\!]}/_x\} \mid \mathsf{parse}_2\ [\![M^\sharp]\!]$ the first test (is_rd) fails, the second one (is_idc) succeeds, yielding two silent transitions. The third test is evaluated by applying the equations that define idc_1,idc_2, idc_3,check:

$$\mathsf{check\_idc}(\mathsf{idc}(a\,,\,v_l\,,\,w_l)) = \mathsf{verify}(v_l\,,\,w_l\,,\,a)\ =\ \mathsf{ok}.$$

The check (and thus also check_idc) test succeeds by construction. In parse_2 , the is_rd test fails, the is_idc test succeeds and reduces into a replicated output on *log*.

$$\begin{aligned}
[\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M^\sharp]\!]\ P\,\{^{[\![M^\sharp]\!]}/_x\}] &\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} [\![\mathcal{C}_a]\!][P\,\{^{[\![M^\sharp]\!]}/_x\} \mid \mathsf{parse}_2\ [\![M]\!]\,]\\
&\xrightarrow{\tau}\xrightarrow{\tau} [\![\mathcal{C}_a]\!][P\,\{^{[\![M^\sharp]\!]}/_x\} \mid \mathsf{repl}\ log!\langle[\![M^\sharp]\!]\rangle].
\end{aligned}$$

Now since the translation of $l.C_l$ within $\mathcal{C}_a$ contains the process $\mathsf{repl}\ log!\langle\mathsf{idc}(a\,,\,v_l\,,\,w_l)\rangle$, we can apply the equivalence Repl_repl:

$$\mathsf{repl}\ log!\langle[\![M^\sharp]\!]\rangle \mid \mathsf{repl}\ log!\langle\mathsf{idc}(a\,,\,v_l\,,\,w_l)\rangle \equiv \mathsf{repl}\ log!\langle\mathsf{idc}(a\,,\,v_l\,,\,w_l)\rangle$$

and obtain

$$[\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M^\sharp]\!]\ P] \xrightarrow{\tau}{}^{*}\ [\![\mathcal{C}_a]\!][P \mid \mathsf{repl}\ log!\langle[\![M^\sharp]\!]\rangle] \equiv [\![\mathcal{C}_a]\!][P]$$

(4) If $M = l\,.\,\mathbf{Idc}\,(e_0\ H\ V)$, then $e0 \notin \mathcal{A}$, $\mathsf{locs}(l, M) = \mathbf{Idc}\,(e_0\ H\ V)$, and $M^\sharp = l.\mathbf{Idc}\,(e_0\,)$. Only rule In applies to the assumption $C_l \xrightarrow{?\mathbf{Idc}\,(e_0\ H\ V)} C'_l$, so necessarily $C'_l = C_l \curlyvee \mathbf{Idc}\,(e_0\ H\ V)$. Two cases depending on C_l are possible.

- If $\mathbf{Idc}\,(e_0\ H\ V) \preceq C_l$ then the translation of C_l must contain active substitutions s_l, v_l, w_l such that $[\![M^\sharp]\!] = \mathsf{idc}(p\,,\ v_l\,,\ w_l)$. Then no update is done and *parse* reduces exactly as in case $M = \mathbf{Idc}\,(a\,)$.
- Otherwise $C_l \preceq \mathbf{Idc}\,(e_0\ H\ V)$, the environment is sending new data.

$$[\![M]\!] = \mathsf{idc}(e_0\,,\ H\,,\ \mathsf{sign}(H\,,\ \mathsf{sk}(m_{e_0}))).$$

In parse_1 , the first is_rd test fails, is_idc and check_idc tests succeed (by translation); in parse_2 , the same is_rd test fails and is_idc test succeeds, and produces a replicated output on *log*:

$$\mathsf{parse}_1\,[\![M^\sharp]\!]\ P\,\{^{[\![M^\sharp]\!]}/_x\} \xrightarrow{\tau}{}^*\ P\,\{^{[\![M^\sharp]\!]}/_x\}\,|\,\mathsf{parse}_2\,[\![M^\sharp]\!] \xrightarrow{\tau}{}^*\ P\,\{^{[\![M^\sharp]\!]}/_x\}\,|\,\mathsf{repl}\,log!\langle[\![M^\sharp]\!]\rangle = P'.$$

We compute the update for location l, by applying structural equivalences to P' and the concerned part of the context $[\![l\,.\,C_l]\!]\,|\,P'$.

Here are different cases depending on C_l.

– If $C_l = \mathbf{Idu}\,(e_0\,H')$ satisfying the definition of f_u function, that is $\mathsf{h}(e_0 + (+_1\,H)) = H'$, we have

$$\begin{aligned}
&[\![l\,.\,\mathbf{Idu}\,(e_0\,H')]\!]\,|\,P' && \text{(B.2.14)}\\
&\equiv \{H'/l\}\,|\,(\{H/v_l\}\,|\,\{\mathsf{sign}(v_l\,,\ \mathsf{sk}(m_{e_0}))/w_l\}\,|\,P') && \text{(B.2.15)}\\
&\equiv \{H'/l\}\,|\,(\varsigma((+_1\,H),(+_2\,H))_{e_0}\,|\,P\,\{^{\mathsf{idc}(e_0\,,\,v_l\,,\,w_l)}/_x\}) && \text{(B.2.16)}\\
&\equiv \varphi((+_1\,H),(+_2\,H))_{e_0}|P\{^{[\![l\,.\,\mathbf{Idc}\,(e_0\,)]\!]}/_x\} && \text{(B.2.17)}\\
&\equiv [\![l\,.\,\mathbf{Idc}\,(e_0\ H\ V)]\!]|P\{^{[\![l\,.\,\mathbf{Idc}\,(e_0\,)]\!]}/_x\}\\
&\equiv [\![l\,.\,C'_l]\!]|P\{^{[\![M^\sharp]\!]}/_x\}
\end{aligned}$$

In B.2.14 we introduce fresh active substitutions for v_l, w_l and we fold them, as well as the replicated logging, into the definition of ς; in B.2.16 we fold the definition of φ; in B.2.17 we fold the translation of the updated location.

We have indeed $l.C'_l = l\,.\,\mathbf{Idc}\,(e_0\ H\ V) = l.\mathbf{Idu}\,(e_0\,H') \curlyvee \mathbf{Idc}\,(e_0\ H\ V) = l.C_l \curlyvee \mathsf{locs}(l, M)$.

– If $C_l = \bot$ then we have

$$\begin{aligned}
&[\![l\,.\,\bot]\!]\,|\,P' && \text{(B.2.18)}\\
&\equiv \varphi((+_1\,H),(+_2\,H))_{e_0}\,|\,P\,\{^{\mathsf{idc}(e_0\,,\,v_l\,,\,w_l)}/_x\}\\
&\equiv [\![l\,.\,\mathbf{Idc}\,(e_0\ H\ V)]\!]\,|\,P\,\{^{[\![M^\sharp]\!]}/_x\}
\end{aligned}$$

In B.2.18 we introduce fresh substitutions to represent location state of l, including the replicated log entry, using φ. Then we fold the translation of the location. We have indeed $l.C'_l = l\,.\,\mathbf{Idc}\,(e_0\ H\ V) = l.\bot \curlyvee \mathbf{Idc}\,(e_0\ H\ V) = l.C_l \curlyvee \mathsf{locs}(l, M)$.

For locations l_0 other than l, $\mathsf{locs}(l_0, M) = \bot$, so $C'_{l_0} = C_{l_0}$.

In both cases we thus have $[\![\mathcal{C}_a]\!][P'] \equiv [\![\mathcal{C}'_a]\!][P\,\{^{[\![M^\sharp]\!]}/_x\}]$.

(5) If $M = l\,.\,\mathbf{Idu}$ or $M = l\,.\,\mathbf{Idu}\,(e\,H)$ then its translation $[\![M^\sharp]\!]$ is $Idu(l)$ or $\mathsf{idu}(H)$ respectively. The first two tests in parse_1 fail, the third one (is_idu) succeeds, so we have

$$[\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M^\sharp]\!]\ P\,\{^{[\![M^\sharp]\!]}/_x\}] \xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} [\![\mathcal{C}_a]\!][\mathsf{parse_c}\,[\![M^\sharp]\!]\,(P\,\{^{[\![M^\sharp]\!]}/_x\}\,|\,\mathsf{parse}_2\,[\![M^\sharp]\!]\,)]$$

.

Since M is committable, by Lemma B.4, we have

$$[\![\mathcal{C}_a]\!][\mathsf{parse_c}\,[\![M^\sharp]\!]\,(P\,\{^{[\![M^\sharp]\!]}/_x\}\mid \mathsf{parse}_2\,[\![M^\sharp]\!]\,)] \xrightarrow{\tau}{}^{*}\ [\![\mathcal{C}'_a]\!][P\,\{^{[\![M^\sharp]\!]}/_x\}\mid \mathsf{parse}_2\,[\![M^\sharp]\!]\,]$$

All the three tests in parse_2 fail so it reduces to 0, thus

$$[\![\mathcal{C}'_a]\!][P\,\{^{[\![M^\sharp]\!]}/_x\}\mid \mathsf{parse}_2\,[\![M^\sharp]\!]\,] \xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} [\![\mathcal{C}'_a]\!][P\,\{^{[\![M^\sharp]\!]}/_x\}]$$

(6) If $M = p$ where p is a name of a principal, Lemma B.4 applies as for the case Idu.

(7) If M is a pair $M_1 + M_2$, then $[\![M]\!] = [\![M_1]\!] + [\![M_2]\!]$. The first three tests of parse_1 fail, the fourth $\mathsf{is\_pair}$ succeeds:

$$[\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M]\!]\,P] \xrightarrow{\tau}{}^{*}\ [\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M_1]\!]\,(\mathsf{parse}_1\,[\![M_2]\!]\,P\mid \mathsf{parse}_2\,[\![M]\!]\,)]$$

By decomposing the assumption, we know that for all $l \in \mathcal{L}$ both $C_l \curlyvee \mathsf{locs}(l, M_1)$ and $C_l \curlyvee \mathsf{locs}(l, M_2)$ exist. We first apply the induction hypothesis on M_1 :

$$[\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M_1]\!]\,(\mathsf{parse}_1\,[\![M_2]\!]\,P\mid \mathsf{parse}_2\,[\![M]\!]\,)] \xrightarrow{\tau}{}^{*}\ [\![\mathcal{C}'_a]\!][\mathsf{parse}_1\,[\![M_2]\!]\,(P\mid \mathsf{parse}_2\,[\![M]\!]\,)]$$

By Lemma B.5(1),

$$[\![\mathcal{C}_a]\!][\mathsf{parse}_1\,[\![M_1]\!]\,(\mathsf{parse}_1\,[\![M_2]\!]\,(P\mid \mathsf{parse}_2\,[\![M]\!]\,))] \xrightarrow{\tau}{}^{*}\ [\![\mathcal{C}_a]\!][\mathsf{parse}_1\,[\![M_2]\!]\,(P\mid \mathsf{parse}_2\,[\![M]\!]\,)]$$

By induction hypothesis on M_2 and by Lemma B.5(1),

$$[\![\mathcal{C}_a]\!][\mathsf{parse}_1\,[\![M_2]\!]\,(P\mid \mathsf{parse}_2\,[\![M]\!]\,)] \longrightarrow_D^{*}\ [\![\mathcal{C}_a]\!][P\mid \mathsf{parse}_2\,[\![M]\!]\,].$$

The first two tests of parse_2 fail, the third $\mathsf{is\_pair}$ succeeds:

$$[\![\mathcal{C}_a]\!][P\mid \mathsf{parse}_2\,[\![M]\!]\,] \longrightarrow_D^{*}\ [\![\mathcal{C}_a]\!][P\mid \mathsf{parse}_2\,[\![M_1]\!]\mid \mathsf{parse}_2\,[\![M_2]\!]\,]$$

Applying Lemma B.5(2) to M_1 and M_2 we get,

$$[\![\mathcal{C}_a]\!][P\mid \mathsf{parse}_2\,[\![M]\!]\,] \longrightarrow_D^{*}\ [\![\mathcal{C}'_a]\!][P\mid 0]$$

By transitivity,

$$[\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M]\!]\,P] \xrightarrow{\tau}{}^{*}\ [\![\mathcal{C}_a]\!][P]$$

(8) The reasoning is the same for other source constructors. □

The parse function enables blaming a principal who owns a received capability that is not compatible with the context.

Lemma B.7 (Accusing a principal)**.** *For any two capabilities $l.C = l.\mathbf{Idc}\,ct$ and $l.C' = l.\mathbf{Idc}\,ct'$ such that $\mathsf{f}_u\,(ct) = \mathsf{f}_u\,(ct')$ and neither $C \preceq C'$ nor $C' \preceq C$, we have*

$$Q\,(\,[\![l\,.\,C]\!]\,)\mid \mathsf{repl}\ log!\langle[\![l\,.\,C']\!]\rangle \Downarrow \mathsf{prin\_of}(C)$$

Proof: Let $C = \mathbf{Idc}\,(e\ H\ V)$ and $C' = \mathbf{Idc}\,(e'\ H'\ V')$. The translation of the **Idc** capabilities is as follows: $l.\mathbf{Idc}\,(e\ H\ V) = \mathsf{idc}(\mathsf{pk}(m_e)\,,\ H\,,\ \mathsf{sign}(H\,,\ \mathsf{sk}(m_e)))$
and $l.\mathbf{Idc}\,(e\ H'\ V') = \mathsf{idc}(\mathsf{pk}(m_e)\,,\ H'\,,\ \mathsf{sign}(H'\,,\ \mathsf{sk}(m_e)))$.
We recall the definition of $Q\,(\,y_1\,)$.

$$\begin{aligned} Q\,(\,y_1\,) \stackrel{\text{def}}{=}\quad & log?(y_2).\mathsf{if}\ \mathsf{check\_idc}(y_1)\ \mathsf{and}\ \mathsf{check\_idc}(y_2)\ \mathsf{then} \\ & \mathsf{if}\ \mathsf{get\_idu}(y_1)\ =\ \mathsf{get\_idu}(y_2)\ \mathsf{and}\ \mathsf{idc}_2(y_1) \neq \mathsf{idc}_2(y_2)\ \mathsf{then}\ bad!\langle\mathsf{get\_prin}(y_1)\rangle \end{aligned}$$

The continuation $Q\,(\,[\![\,l\,.\,C\,]\!]\,)$ communicates with the replicated output on log. We have

$$\begin{array}{l}\mathsf{if\ check\_idc}([\![\,l\,.\,C\,]\!])\ \mathsf{and\ check\_idc}([\![\,l\,.\,C'\,]\!])\ \mathsf{then}\\ \quad\mathsf{if\ get\_idu}([\![\,l\,.\,C\,]\!])\ =\ \mathsf{get\_idu}([\![\,l\,.\,C'\,]\!])\ \mathsf{and}\ \mathsf{idc}_2([\![\,l\,.\,C\,]\!]) \neq \mathsf{idc}_2([\![\,l\,.\,C'\,]\!])\\ \quad\mathsf{then}\ bad!\langle\mathsf{get\_prin}([\![\,l\,.\,C\,]\!])\rangle \mid \mathsf{repl}\ log!\langle[\![\,l\,.\,C'\,]\!]\rangle\end{array}$$

The resolution continuation can now reduce. By assumption, the Idcs are valid, so the first idc-integrity tests succeed. We apply the equations defining $\mathsf{get\_idu}$ and idc_2.

The assumption $\mathsf{f}_u\,(ct) = \mathsf{f}_u\,(ct')$ implies that $e = e'$ and $\mathsf{h}(\mathsf{pk}(m_e) + (+_1\,H)) = \mathsf{h}(\mathsf{pk}(m_{e'}) + (+_1\,H'))$, and since $\mathbf{h}$ is injective, we have $(+_1\,H) = (+_1\,H')$. So the Idus comparison succeeds:

$$\begin{array}{l}\mathsf{get\_idu}(\mathsf{idc}(\mathsf{pk}(m_e)\,,\,H\,,\,\mathsf{sign}(H\,,\,\mathsf{sk}(m_e)))) = \mathsf{idu}(\mathsf{h}(\mathsf{pk}(m_e) + (+_1\,H)))\\ = \mathsf{idu}(\mathsf{h}(\mathsf{pk}(m_{e'}) + (+_1\,H'))) = \mathsf{get\_idu}(\mathsf{idc}(\mathsf{pk}(m_{e'})\,,\,H'\,,\,\mathsf{sign}(H\,,\,\mathsf{sk}(m_{e'}))))\end{array}$$

Another assumption says that neither $C \preceq C'$ nor $C' \preceq C$, which implies $H \neq H'$ and $V \neq V'$. Thus the capabilities contain different values:

$$\begin{array}{l}\mathsf{idc}_2(\mathsf{idc}(\mathsf{pk}(m_e)\,,\,H\,,\,\mathsf{sign}(H\,,\,\mathsf{sk}(m_e)))) = H\\ \mathsf{idc}_2(\mathsf{idc}(\mathsf{pk}(m_e)\,,\,H'\,,\,\mathsf{sign}(H'\,,\,\mathsf{sk}(m_e)))) = H'\end{array}$$

All tests within Q succeed, so after three silent transitions, the output on bad is enabled:

$$Q\,(\,[\![\,l\,.\,C\,]\!]\,) \mid \mathsf{repl}\ log!\langle[\![\,l\,.\,C'\,]\!]\rangle \xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{bad!e} \mathsf{repl}\ log!\langle[\![\,l\,.\,C'\,]\!]\rangle$$

Indeed, we have $\mathsf{get\_prin}(\mathsf{idc}(\mathsf{pk}(m_e)\,,\,v_l\,,\,w_l)) = \mathsf{pk}(m_e)$ that is indeed the translation of $\mathsf{prin\_of}(C)$. □

Definition B.2.3 (Related capabilities)**.** Two source capabilities $l.C_1$ and $l.C_2$ are related if for all ct_1, ct_2 such that $\mathbf{Idu}\,ct_1 \preceq C_1$ and $\mathbf{Idu}\,ct_2 \preceq C_2$, we have $ct_1 = ct_2$.

Lemma B.8 (Parsing an incompatible term)**.** *For any well-formed source context $\mathcal{C}_a$,*

$$\mathcal{C}_a[\_] = \nu\mathcal{N}.\left(\textstyle\prod_{l\in\mathcal{L}} l.C_l \mid \prod_{a'\in\mathcal{A}\setminus\{a\}} a'[P_{a'}] \mid a[\_] \mid \phi \mid \mathcal{T}\right)$$

for any marshallable source term M, for any translation of a process P, if there is a location $l \in \mathcal{L}$ such that C_l and $\mathsf{locs}(l, M)$ are related but their sup does not exist, then

$$[\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M]\!]\,P] \Downarrow\ \mathsf{prin\_of}(\mathsf{locs}(l, M))\ \textit{and}\ \mathsf{prin\_of}(\mathsf{locs}(l, M)) \notin \mathcal{A}.$$

Proof: Let $\mathcal{C}_a$ be a well-formed source context. Let $l \in \mathcal{L}$ be a location, such that neither $C_l \preceq \mathsf{locs}(l, M)$ nor $\mathsf{locs}(l, M) \preceq C_l$.

We proceed by structural induction on terms M such that $\mathsf{locs}(l, M) \neq \bot$, namely on capabilities and pairs.

- If $M = l\,.\,\mathbf{Rd}\,(e'\,H'\,V')$, then the faulty capability is necessarily l. Indeed, M is well-sorted, and thus it may only contain **Idu** capability which is comparable to any capability in our order. **Rd** capabilities are not comparable to **Rd**s and **Idc**s containing value $V' \neq V$. In parse_1 code, and parse_2, tests $\mathsf{is\_rdc}$ and $\mathsf{check\_idc}$ on the received value $[\![l\,.\,\mathbf{Rd}\,(e'\,H'\,V')]\!]$ succeed: $\mathsf{parse\,rd}(e'\,,\,H'\,,\,[\![\,V'\,]\!]\,,\,\mathsf{sign}(\mathsf{h}(H') + \mathsf{h}(H' + [\![\,V'\,]\!])\,,\,\mathsf{sk}(m_{e'})))\ P \xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} P \mid R_3$ where $R_3 = \mathsf{repl}\ log!\langle\mathsf{idc}(e'\,,\,\mathsf{h}(H') + \mathsf{h}(H' + [\![\,V'\,]\!])\,,\,\mathsf{sign}(\mathsf{h}(H') + \mathsf{h}(H' + [\![\,V'\,]\!])\,,\,\mathsf{sk}(m_{e'})))\rangle$
  $= \mathsf{repl}\ log!\langle[\![\,l\,.\,\mathbf{Idc}\,(e'\,\mathsf{h}(H') + \mathsf{h}(H' + [\![\,V'\,]\!])\,V')]\!]\rangle$.
  We now consider the interactions of R_3 and of the resolution process R which is always available in the translation.

We recall the definition of the resolution process:

$$Q\,(\,y_1\,) \stackrel{\text{def}}{=} \quad log?(y_2).\mathsf{if}\ \mathsf{check\_idc}(y_1)\ \mathsf{and}\ \mathsf{check\_idc}(y_2)\ \mathsf{then}\ \mathsf{if}\ \mathsf{get\_idu}(y_1) = \mathsf{get\_idu}(y_2)\ \mathsf{and}\ \mathsf{idc}_2(y_1) \neq \mathsf{idc}_2(y_2)\ \mathsf{then}\ bad!\langle\mathsf{get\_prin}(y_1)\rangle$$

$$R \stackrel{\text{def}}{=} (\mathsf{repl}\ log?(y_1).Q\,(\,y_1\,)) \mid (\mathsf{repl}\ log!\langle\mathsf{None}\rangle)$$

Process R can receive a first **Idc** provided by R_2. We have

$$R_3 \mid R \longrightarrow R_3 \mid R \mid Q([\![l\,.\,\mathbf{Idc}\,(e'\,\mathbf{h}(H')+\mathbf{h}(H'+[\![\,V'\,]\!]\ V')]\!])$$

– Suppose that $C_l = \mathbf{Rd}\,(e\ H\ V)$ and $V' \neq V$.
By assumption, $l.\mathsf{locs}(l, M) = l.\mathbf{Rd}\,(e'\ H'\ V')$ and $l.C_l = l.\mathbf{Rd}\,(e\ H\ V)$ are related. We use the ordering rule Idc, then Idu and transitivity to derive $\mathbf{Idu}\,\mathsf{f}_u\,(\mathsf{f}_c\,(e'\ H'\ V')) \preceq \mathbf{Rd}\,(e'\ H'\ V')$ and $\mathbf{Idu}\,\mathsf{f}_u\,(\mathsf{f}_c\,(e\ H\ V)) \preceq \mathbf{Rd}\,(e\ H\ V)$.
By definition of related capabilities, $\mathbf{Idu}\,\mathsf{f}_u\,(\mathsf{f}_c\,(e'\ H'\ V')) = \mathbf{Idu}\,\mathsf{f}_u\,(\mathsf{f}_c\,(e\ H\ V))$. We derive the equality $\mathsf{f}_u\,(\mathsf{f}_c\,(e'\ H'\ V')) = \mathsf{f}_u\,(\mathsf{f}_c\,(e\ H\ V))$.
The translation $[\![l\,.\,C_l]\!]$ contains the process

$$R_2 = \mathsf{repl}\ log!\langle[\![l\,.\,\mathbf{Idc}\,(e\,\mathsf{h}(H)+\mathsf{h}(H+[\![\,V\,]\!]\ V)]\!]\rangle$$

We can apply Lemma B.7 and get

$$R_2 \mid Q([\![l\,.\,\mathbf{Idc}\,(e'\,\mathbf{h}(H')+\mathbf{h}(H'+[\![\,V'\,]\!]\ V')]\!] \Downarrow e$$

– Suppose $C_l = \mathbf{Idc}\,(e\ H\ V)$ and $V \neq V'$.
By assumption, $l.\mathsf{locs}(l, M) = l.\mathbf{Rd}\,(e'\ H'\ V')$ and $l.C_l = l.\mathbf{Idc}\,(e\ H\ V)$ are related. and $\mathbf{Idu}\,ct' \preceq C_l$. Using the ordering rules Idc, Idu and transitivity, we get
$\mathbf{Idu}\,\mathsf{f}_u\,(\mathsf{f}_c\,(e'\ H'\ V')) \preceq \mathbf{Rd}\,(e'\ H'\ V')$ and $\mathbf{Idu}\,\mathsf{f}_u\,(e\ H\ V) \preceq \mathbf{Idc}\,(e\ H\ V)$.
By definition of related capabilities, $\mathbf{Idu}\,\mathsf{f}_u\,(\mathsf{f}_c\,(e'\ H'\ V')) = \mathbf{Idu}\,\mathsf{f}_u\,(e\ H\ V)$. We derive the equality $\mathsf{f}_u\,(\mathsf{f}_c\,(e'\ H'\ V')) = \mathsf{f}_u\,(e\ H\ V)$.
The translation of $[\![l\,.\,C_l]\!]$ contains the process

$$R_2 = \mathsf{repl}\ log!\langle[\![l\,.\,\mathbf{Idc}\,(e\ H\ V)]\!]\rangle$$

We can apply Lemma B.7 and get

$$R_2 \mid Q([\![l\,.\,\mathbf{Idc}\,(e'\,\mathbf{h}(H')+\mathbf{h}(H'+[\![\,V'\,]\!]\ V')]\!] \Downarrow e$$

– If $M = l\,.\,\mathbf{Idc}\,(e\ M_1+M_2\ V)$, then the faulty capability is necessarily l. As in the previous case, there are two possibles values for C_l : $C_l = \mathbf{Idc}\,(e\ H'\ V')$ or $C_l = \mathbf{Rd}\,(e\ H'\ V')$.
– If $M = l\,.\,\mathbf{Idu}$, the hypothesis does not hold as $l\,.\,\mathbf{Idu}$ is comparable to any capability in our order.
– If $M = M_1+M_2$, then after four transitions (four failures + a success), the process makes recursive calls of parse on subterms:

$$[\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M_1 + M_2]\!]\ P] \xrightarrow{\tau}{}^{*}\ [\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M_1]\!]\ \mathsf{parse}\,[\![M_2]\!]\ P].$$

We can apply the induction hypotheses to the subterms M_1 and M_2. If the faulty capability l is in M_1, then $[\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M_1]\!]\ P'] \Downarrow e$. Otherwise $P' = [\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M_2]\!]\ P] \Downarrow e$.
□

Lemma B.9 (Send updates)**.** *For a any target process P within a translation of a well-formed source context $\mathcal{C}_a$, Let $\mathcal{D}_a$ be a source context where x is bound in the hole:*

$$\mathcal{D}_a[\_] = \textstyle\prod_{a' \in \mathcal{A}\setminus\{a\}} a'[P_{a'}] \mid a[\_] \mid \phi \mid \mathcal{T}$$

Any marshallable source term M can be sent from a well-formed source context, so that the resulting context is also well-formed:

$$\nu\mathcal{N}.[\![(\textstyle\prod_{l\in\mathcal{L}} l.C_l \mid \mathcal{D}_a)]\!]\,[\,c!\langle M\rangle.P] \xrightarrow{\nu\tilde{u}.\,c\,!<M>} \nu\mathcal{N}\setminus\tilde{u}.[\![(\textstyle\prod_{l\in\mathcal{L}} l.(C_l \curlyvee \mathsf{locs}(l, M)) \mid \mathcal{D}_a)]\!].$$

PROOF: We proceed by induction on the structure of M.

- $M = p$, a name of a principal. We have $[\![p]\!] = p$. Names of principals are not restricted; they are translated into homonymous variables bound with toplevel active substitutions, thus $[\![\mathcal{C}_a]\!][c!\langle p\rangle.P] \xrightarrow{c!p} [\![\mathcal{C}_a]\!][P]$.
- $M = M_1 + M_2$, a constructed value.
  We have $[\![M]\!] = [\![M_1]\!] + [\![M_2]\!]$. Applying the induction hypothesis on M_1 and M_2 we have

$$\nu\mathcal{N}.[\![(\textstyle\prod_{l\in\mathcal{L}} l.C_l \mid \mathcal{D}_a)]\!]\,[\,c!\langle M_1\rangle.P] \xrightarrow{\nu\tilde{u_1}.\, c!<M_1>} \nu\mathcal{N}\setminus\tilde{u_1}.[\![(\textstyle\prod_{l\in\mathcal{L}} l.(C_l \curlyvee \mathsf{locs}(l, M_1)) \mid \mathcal{D}_a)]\!]$$

$$\nu\mathcal{N}.[\![(\textstyle\prod_{l\in\mathcal{L}} l.C_l \mid \mathcal{D}_a)]\!]\,[\,c!\langle M_2\rangle.P] \xrightarrow{\nu\tilde{u_2}.\, c!<M_2>} \nu\mathcal{N}\setminus\tilde{u_2}.[\![(\textstyle\prod_{l\in\mathcal{L}} l.(C_l \curlyvee \mathsf{locs}(l, M_2)) \mid \mathcal{D}_a)]\!]$$

  Let $\tilde{u} = \tilde{u_1} \cup \tilde{u_2}$. By definition, $\mathsf{locs}(l, M_1+M_2) = \mathsf{locs}(l, M_1) \curlyvee \mathsf{locs}(l, M_2)$ so

$$\nu\mathcal{N}.[\![(\textstyle\prod_{l\in\mathcal{L}} l.C_l \mid \mathcal{D}_a)]\!]\,[\,c!\langle M\rangle.P] \xrightarrow{\nu\tilde{u}.\, c!<M>} \nu\mathcal{N}\setminus\tilde{u}.[\![(\textstyle\prod_{l\in\mathcal{L}} l.(C_l \curlyvee \mathsf{locs}(l, M)) \mid \mathcal{D}_a)]\!]$$

- $M = l.Cap\,ct$ capability
  For all capabilities except Rd, $\mathsf{locs}(l, l\,.\,Cap\ ct) = Cap\ ct$, and $\mathsf{locs}(l', l\,.\,Cap\ ct) = \bot$ for $l' \neq l$. We have

$$[\![l\,.\,C_l]\!]|c!\langle[\![l\,.\,Cap\ ct]\!]\rangle.P \xrightarrow{\nu\tilde{u}.\, c![\![l\,.\,Cap\ ct]\!]} [\![l\,.\,C_l \curlyvee Cap\ ct]\!]|\,P$$

  Since the translation of $l.Cap(ct)$ is the same for all Cap, $[\![l\,.\,C_l]\!] = [\![l\,.\,(C_l \curlyvee Cap\ ct)]\!] = [\![l\,.\,Cap\ ct]\!] = [\![l\,.\,C'_l]\!]$. (Note that if $C_l \preceq Cap\ ct$ then necessarily $\mathsf{prin\_of}(C_l) \in \mathcal{A}$.)
  The table D contains the label restrictions $\tilde{u}$ given a C_l and an output $\nu\,\tilde{u}\,.\,c!\,M$(in red what is not disclosed directly, but can derived). Column "exported" recalls what variables are not restricted on top of C_l by translation. Empty set is denoted with -. As in source rule

$$\frac{}{C \xrightarrow{!\,C'} C \curlyvee C'}\ \textsc{Out}$$

  in the translation we have $\mathrm{D}[C_l][\text{exported}] \cup \mathrm{D}[C_l][C] = \mathrm{D}[\text{exported}][C_l \curlyvee C]$

$C_l \backslash^C$	exported	**Idu**	**Idc** $(p\ )$	**Rd** $(p\ \ V)$
$\bot$	-	l	l,v_l, w_l	l, v_l,w_l, s_l
Idu	l	-	v_l, w_l	v_l,w_l, s_l
Idc $(p\ H\ V)$	l, v_l, w_l	-	-	s_l
Rd $(p\ H\ V)$	l, v_l, w_l, s_l	-	-	-

$$[\![(\textstyle\prod_{l\in\mathcal{L}} l.C_l \mid \mathcal{D}_a)]\!]\,[\,c!\langle M\rangle.P] \xrightarrow{\nu\tilde{u}.\, c![\![M]\!]} \nu\mathcal{N}\setminus\tilde{u}.[\![(\textstyle\prod_{l\in\mathcal{L}} l.(C_l \curlyvee \mathsf{locs}(l, M)) \mid \mathcal{D}_a)]\!]$$

  If $M = l.\mathbf{Rd}\,(a_0\ \ V)$ capability, then we should also apply the induction hypothesis on V:

$$[\![(\textstyle\prod_{l\in\mathcal{L}} l.C_l \mid \mathcal{D}_a)]\!]\,[\,c!\langle V\rangle.P] \xrightarrow{\nu\tilde{u_1}.\, c!<V>} \nu\mathcal{N}\setminus\tilde{u_1}.[\![(\textstyle\prod_{l\in\mathcal{L}} l.(C_l \curlyvee \mathsf{locs}(l, V)) \mid \mathcal{D}_a)]\!]$$

  For this case, we have $u = u_1 \cup$ Disclosed$[C_l \curlyvee \mathsf{locs}(l, V)][\mathbf{Rd}\,(a_0\ \ V)]$.

$$[\![(\textstyle\prod_{l\in\mathcal{L}} l.C_l \mid \mathcal{D}_a)]\!]\,[\,c!\langle M\rangle.P] \xrightarrow{\nu\tilde{u}.\, c![\![l\,.\,\mathbf{Rd}\,(a_0\ \ V)]\!]} \nu\mathcal{N}\setminus\tilde{u}.[\![(\textstyle\prod_{l\in\mathcal{L}} l.(C_l \curlyvee \mathsf{locs}(l, M)) \mid \mathcal{D}_a)]\!]$$

□

Lemma B.10 (Equality of translated terms)**.** *For all source terms M_1 and M_2 within a well-formed source context, we have $M_1 = M_2$ if and only if $[\![M_1]\!] = [\![M_2]\!]$.*

PROOF: By case analysis on the source rule used to derive $M_1 = M_2$.

- ATOM
  If for some name or variable u, $M_1 = M_2 = u$, then trivially $[\![M_1]\!] = [\![M_2]\!] = u$. If $M_1 = u$ and $M_2 = v$ with $u \neq v$ then $[\![M_1]\!] = u$, $[\![M_2]\!] = v$, and so $[\![M_1]\!] \neq [\![M_2]\!]$.
- FST
  Let $M_1 = +_1((M_2 + M_3))$. Then $[\![M_1]\!] = (+_1 [\![M_2]\!] + [\![M_3]\!])$. By definition of target projections, we have $[\![M_1]\!] = [\![M_2]\!]$. In the same way, $M_1 \neq M_2$ implies $[\![M_1]\!] \neq [\![M_2]\!]$. Similar for the case SND.
- IDU_OF_IDC
  Let $M_1 = \mathsf{get\_idu}(x\,.\,\mathbf{Idc}\,(p\,))$ and $M_2 = x\,.\,\mathbf{Idu}$. We have

  $$[\![M_1]\!] = \mathsf{get\_idu}(\mathsf{idc}(p\,,\,v_x\,,\,w_x)) = \mathsf{idu}(\mathsf{h}(p + (+_1\,v_x))) \text{ and } [\![M_2]\!] = \mathsf{idu}(x).$$

  Since the translation of the context is well-formed, it provides active substitutions $\{\mathsf{h}(p + M_1')/x\}$ and $\{M_1' + M_2'/v_x\}$ for some M_1', M_2'.
  So $[\![M_1]\!] = \mathsf{idu}(\mathsf{h}(p + (+_1\,v_x))) = \mathsf{idu}(\mathsf{h}(p + M_1')) = \mathsf{idu}(x) = [\![M_2]\!]$.
- IDU_OF_RD
  Let $M_1 = \mathsf{get\_idu}(x\,.\,\mathbf{Rd}\,(p\;v))$ and $M_2 = x\,.\,\mathbf{Idu}$. We have

  $$[\![M_1]\!] = \mathsf{get\_idu}(\mathsf{rd}(p\,,\,s_x\,,\,[\![v]\!]\,,\,w_x)) = \mathsf{idu}(\mathsf{h}(p + \mathsf{h}(s_x))), \text{ and } [\![M_2]\!] = \mathsf{idu}(x).$$

  Since the translation of the context is well-formed, it provides an active substitution $\{\mathsf{h}(p + \mathsf{h}(s_x))/x\}$. So $[\![M_1]\!] = \mathsf{idu}(\mathsf{h}(p + \mathsf{h}(s_x))) = \mathsf{idu}(x) = [\![M_2]\!]$.
- IDC_OF_RD
  Let $M_1 = \mathsf{get\_idc}(x\,.\,\mathbf{Rd}\,(p\;v))$ and $M_2 = x\,.\,\mathbf{Idc}\,(p\,)$. We have

  $$[\![M_1]\!] = \mathsf{get\_idc}(\mathsf{rd}(p\,,\,s_x\,,\,[\![v]\!]\,,\,w_x)) = \mathsf{idc}(p\,,\,\mathsf{h}(p)+\mathsf{h}(s_x+[\![v]\!])\,,\,w_x), \text{ and } [\![M_2]\!] = \mathsf{idc}(p\,,\,v_x\,,\,w_x).$$

  Since the translation of the context is well-formed, it provides exactly the active substitution $\{\mathsf{h}(s_x) + \mathsf{h}(s_x + [\![v]\!])/v_x\}$. So $[\![M_1]\!] = \mathsf{idc}(p\,,\,v_x\,,\,w_x) = [\![M_2]\!]$.
- PRIN_OF_IDC
  Let $M_1 = \mathsf{get\_prin}(x\,.\,\mathbf{Idc}\,(p\,))$ and $M_2 = p$. We have $[\![M_1]\!] = \mathsf{get\_prin}(\mathsf{idc}(p\,,\,v_x\,,\,w_x)) = p = [\![M_2]\!]$. Similar for the case PRIN_OF_RD.
- READ
  Let $M_1 = \mathsf{read}(x\,.\,\mathbf{Rd}\,(p\;v))$ and $M_2 = v$. We have $[\![M_1]\!] = \mathsf{read}(\mathsf{rd}(p\,,\,s_x\,,\,[\![v]\!]\,,\,w_x)) = [\![v]\!] = [\![M_2]\!]$.
- TEST_IDU
  Let $M_1 = \mathsf{is\_idu}(x\,.\,\mathbf{Idu})$ and $M_2 = ok$. We have $[\![M_1]\!] = \mathsf{is\_idu}(\mathsf{idu}(x)) = \mathsf{ok} = [\![M_2]\!]$. Similar for the case TEST_RD, TEST_IDC.

□

B.2.2 Functional adequacy

Lemma B.11 (Functional adequacy for one source non-silent step)**.** *Let A be a well-formed source system. For all transition step $A \xrightarrow{\phi} A'$, such that $\phi \neq \tau$, there exists a trace $[\![A]\!] \xrightarrow{\psi}{}^{*} [\![A']\!]$.*

PROOF: Let A be a well-formed source system.

$$A \;\equiv\; \nu\mathcal{N}\,(\textstyle\prod_{l\in\mathcal{L}} l.C_l \mid \prod_{a'\in\mathcal{A}} a'[P_{a'}] \mid \phi \mid \mathcal{T})$$

Let $\mathcal{C}_a$ be the well formed source context built on A where P_a is replaced with a hole:

$$\mathcal{C}_a[\_] = \nu\mathcal{N}.\left(\prod_{l\in\mathcal{L}} l.C_l \mid \prod_{a'\in\mathcal{A}\setminus\{a\}} a'[P_{a'}] \mid a[\_] \mid \phi \mid \mathcal{T}\right)$$

Suppose that for some A', $A \xrightarrow{\phi} A'$. Since A is well-formed, A' is also well-formed. Let ${\mathcal{C}'}_a$ be the well formed source context built on $A' = {\mathcal{C}'}_a[{P'}_a]$:

$${\mathcal{C}'}_a[\_] = \nu\mathcal{N}'.\left(\prod_{l\in\mathcal{L}'} l.C'_l \mid \prod_{a'\in\mathcal{A}\setminus\{a\}} a'[P_{a'}] \mid a[\_] \mid \phi' \mid \mathcal{T}'\right)$$

Since A is well-formed, we have $A \xrightarrow{\phi} {\mathcal{C}'}_a[{P'}_a]$ for some label $\phi \neq \tau$ and for some system A' based on context ${\mathcal{C}'}_a$. We perform a case analysis on the source label ϕ.

Label ϕ is necessarily input or output.

Case Input If A makes a transition labelled with $c\,?\,(\,M\,)$ then for some a, P_1, and P_2 we have $P_a \equiv c?(x).P_1 \mid P_2$ and $P'_a \equiv P_1\{^M/_x\} \mid P_2$.
By definition, the transition $A \xrightarrow{c?M} A'$ is derived by applying first the base rule RECEIVE_TERM

$$\frac{}{a[c?(x).P] \xrightarrow{c?M} a[P\{^{M\sharp}/_x\}]} \quad \text{RECEIVE\_TERM}$$

then the context rule UP_INfor each $l \in \mathcal{L}$:

$$\left(\frac{A \xrightarrow{c?M} A' \wedge C_0 \xrightarrow{?\mathsf{locs}(l,M)} C_1}{l\,.\,C_0 \mid A \xrightarrow{c?M} l\,.\,C_1 \mid A'} \quad \text{UP\_IN}\right) \quad \forall l \in \mathcal{L}$$

To be received by A, all capabilities $l \in \mathcal{L}$ are subject to the following rules, by instantiating C to C_l and C' to $\mathsf{locs}(l, M)$:

$$\frac{C' \preceq C \wedge \mathsf{prin\_of}(C') \in \mathcal{A}}{C \xrightarrow{?C'} C} \quad \text{IN\_OWNED} \qquad \frac{\mathsf{prin\_of}(C') \notin \mathcal{A}}{C \xrightarrow{?C'} C \curlyvee C'} \quad \text{IN}$$

Thus in the resulting context, we have for all $l \in \mathcal{L}$, $C'_l = C_l \curlyvee \mathsf{locs}(l, M)$ (when $\mathsf{locs}(l, M) = \bot$, we have $C'_l = C_l$).

In the translation $[\![P]\!]_a = (\nu\, r\,.\,((c?(x).\mathsf{parse}\, x\, [\![P_1]\!]_a) \mid \mathsf{repl}\, r?(\_).c?(x).\mathsf{parse}\, x\, [\![P_1]\!]_a)) \mid [\![P_2]\!]_a$. Since P_a can receive a source term M, the translation of P_a can receive the translation of M:

$$[\![\mathcal{C}_a]\!][[\![P]\!]_a] \xrightarrow{c\,?\,(\,[\![M]\!]\,)} S \text{ where}$$

$$S = [\![\mathcal{C}_a]\!][(\nu\, r\,.\,((\mathsf{parse}\, [\![M]\!]\, ([\![P_1]\!]_a\, \{^{[\![M]\!]}/_x\}))) \mid \mathsf{repl}\, r?(\_).c?(x).\mathsf{parse}\, [\![M]\!]\, [\![P_1\, \{^{[\![M]\!]}/_x\}]\!]_a)) \mid [\![P_2]\!]_a].$$

Lemma B.6 on compatible inputs applies to the process $\mathsf{parse}\, [\![M]\!]\, ([\![P_1]\!]_a\, \{^{[\![M]\!]}/_x\})$ and reduces into a well-formed source context $\mathcal{C}'_a$ such that

$$[\![\mathcal{C}_a]\!][\mathsf{parse}\, [\![M]\!]\, ([\![P_1]\!]_a\, \{^{[\![M]\!]}/_x\})] \xrightarrow{\tau}{}^{*} [\![{\mathcal{C}'}_a]\!][[\![P_1]\!]_a\, \{^{[\![M]\!]}/_x\}] \equiv [\![{\mathcal{C}'}_a]\!][[\![P_1\, \{^M/_x\}]\!]_a]$$

and for all $l \in \mathcal{L}$, $C'_l = C_l \curlyvee \mathsf{locs}(l, M)$. Applying STRUCT, we also have

$$S \xrightarrow{\tau}{}^{*} [\![{\mathcal{C}'}_a]\!][\nu\, r\,.\,([\![P_1\, \{^M/_x\}]\!]_a \mid \mathsf{repl}\, r?(\_).c?(x).\mathsf{parse}\, [\![M]\!]\, [\![P_1\, \{^M/_x\}]\!]_a \mid [\![P_2]\!]_a)]$$

(Since r is fresh in P_1, we get rid of the replicated input on r using the rule DEAD_LOOP.)

$$\equiv [\![{\mathcal{C}'}_a]\!][[\![P_1\, \{^M/_x\}]\!]_a \mid [\![P_2]\!]_a] \equiv [\![{\mathcal{C}'}_a]\!][[\![P']\!]_a] \equiv [\![A']\!].$$

Case Output If A makes a transition labelled $\nu\tilde{u'}.\, c\,!\,M$ then for some a, P_1, P_2 we have $P_a = c!\langle M\rangle.P_1 \mid P_2$, $P'_a = P_1 \mid P_2$. Note that our sort system limits M to marshallable terms.

By definition, the transition $A \xrightarrow{\nu\,\tilde{u}\,.\,c\,!\,M} A'$ is derived by applying first the base rule SEND_TERM,

$$\frac{}{a[c!\langle M\rangle.P] \xrightarrow{c\,!\,M} a[P]} \quad \textsc{Send\_term}$$

then a context rule UP_OUT for each $l \in \mathcal{L}$

$$\left(\frac{A \xrightarrow{c\,!\,M} A' \wedge C_0 \quad \xrightarrow{!\,\mathsf{locs}(l,M)} C_1}{l\,.\,C_0 \mid A \xrightarrow{c\,!\,M} l\,.\,C_1 \mid A'} \quad \textsc{Up\_out} \right) \quad \forall l \in \mathcal{L}$$

and finally $|\tilde{u'}|$ applications of OPEN:

$$\left(\frac{A \xrightarrow{c\,!\,M} A' \wedge (\, c \neq r \wedge r \in M\,)}{\nu\, r\,.\,A \xrightarrow{\nu\, r\,.\, c\,!\,M} A'} \quad \textsc{Open} \right) \qquad |\tilde{u'}| \text{ times}$$

We apply Lemma B.9 to the translations of the source process $c!\langle M\rangle.P_1$, context $\mathcal{C}_a$ and the source term M. According to this lemma, all source locations and there translations are updated correctly on output. More precisely, there exists a well-formed source context $\mathcal{C}'_a$ such that $[\![\mathcal{C}_a]\!][c!\langle[\![M]\!]\rangle.[\![P_1]\!]_a] \xrightarrow{c![\![M]\!]} [\![\mathcal{C}'_a]\!][[\![P_1]\!]_a]$. Then by applying rule STRUCT we obtain

$$[\![\mathcal{C}_a]\!][(c!\langle[\![M]\!]\rangle.[\![P_1]\!]_a) \mid [\![P_2]\!]_a] \xrightarrow{c![\![M]\!]} [\![\mathcal{C}'_a]\!][[\![P_1 \mid P_2]\!]_a] \equiv [\![\mathcal{C}'_a]\!][[\![P']\!]_a] \equiv A'.$$

□

Lemma B.12 (Functional adequacy for one source silent step)**.** *Let A be a well-formed source system. For all transition step $A \xrightarrow{\tau} A'$, there exists a trace $[\![A]\!] \xrightarrow{\psi}{}^{*} [\![A']\!]$.*

PROOF: Let A be a well-formed source system. Suppose $A \xrightarrow{\tau} A'$ for some label ϕ and for some system A'. A makes silent transitions as a result of some internal reductions. We analyse the rule used for the reduction of A.

Since A is well-formed, and $A \longrightarrow A'$, we have:

$$A \;\equiv\; \nu\mathcal{N}\,(\textstyle\prod_{l\in\mathcal{L}} l.C_l \mid \prod_{a'\in\mathcal{A}} a'[P_{a'}] \mid \phi \mid \mathcal{T})$$

$$A' \;\equiv\; \nu\mathcal{N}'\,(\textstyle\prod_{l\in\mathcal{L}'} l.C'_l \mid \prod_{a'\in\mathcal{A}} a'[P'_{a'}] \mid \phi' \mid \mathcal{T}')$$

Let $\mathcal{C}_a$ be the source context built on A where P_a is replaced with a hole:

$$\mathcal{C}_a[\_] = \nu\mathcal{N}.\,\Big(\textstyle\prod_{l\in\mathcal{L}} l.C_l \mid \prod_{a'\in\mathcal{A}\setminus\{a\}} a'[P_{a'}] \mid a[\_] \mid \phi \mid \mathcal{T}\Big)$$

And let $\mathcal{C}'_a$ be the source context built on A':

$$\mathcal{C}'_a[\_] = \nu\mathcal{N}'.\,\Big(\textstyle\prod_{l\in\mathcal{L}'} l.C'_l \mid \prod_{a'\in\mathcal{A}\setminus\{a\}} a'[P_{a'}] \mid a[\_] \mid \phi' \mid \mathcal{T}'\Big)$$

Case newloc

If A reduces by rule NEWLOC then for some a, P_1, P_2 we have $P_a \equiv \mathsf{newloc}\,(x, y).P_1 \mid P_2$, $P'_a = P_1\{^{l}/_{x}\}\{^{l\,.\,\mathbf{Idu}}/_{y}\} \mid P_2$. Lemma B.2 applies with $\phi' = \phi$, $\mathcal{T}' = \mathcal{T}$, $\mathcal{N}' = \mathcal{N} \cup l$, $\mathcal{L}' = \mathcal{L} \cup l$ and the corresponding location $l.C'_l = l.0\,(a\,)$.

Case commit

If A reduces by rule COMMIT_EXT then for some a, P_1, P_2 we have $P_a = \mathsf{commit}\ V\ l\,(x').P_1 \mid P_2$ and $P'_a = P_1\{^{l\,.\,\mathbf{Rd}\,(a\ \ V)}/_{x'}\} \mid P_2$ such that $C_l = l.Cap\,(a\,)$, $C'_l = l.Cap\,(a\ \ V)$.

$$\frac{Cap\ =\ 0\ \vee\ Cap\ =\ \mathbf{Idu}}{l\,.\,Cap\,(a\,)\ \mid\ a[\mathsf{commit}\,M\,l\,(x).P]\ \longrightarrow\ l\,.\,Cap\,(a\ \ M)\ \mid\ a[P\{^{l\,.\,\mathbf{Rd}\,(a\ \ M)}/_{x}\}]}\ \text{COMMIT\_EXT}$$

Well-formedness condition 1) implies $Cap \in \{\mathbf{Idu}, \mathbf{0}\}$ (at most **Idu**) has been exported for location l). The translation for both possible capabilities is $[\![l\,.\,Cap\,(a\,)]\!] = [\![l\,.\,Cap\,(a\ \ V)]\!] = c_l!\langle\mathsf{None}\rangle \mid \{\mathsf{h}(a + \mathsf{h}(s_l))/l\} \mid \nu\, s\,.\,\{s/s_l\}$, with the following toplevel restrictions: l, s_l for $\bot$ and only s_l **Idu**.

By Lemma B.3, we have

$$[\![\mathsf{commit}\ V\ l\,(x').P_1]\!]_a = c_l?(\_).\nu\, v_l\,.\,\nu\, w_l\,.\,(\varsigma(\mathsf{h}(s_l), \mathsf{h}(s_l + [\![V]\!]))_a \mid [\![P_1]\!]_a\,\{^{\mathsf{rd}(a\,,\,s_l\,,\,[\![V]\!]\,,\,w_l)}/_{x'}\})$$

The translations $[\![P]\!]_a \mid [\![l\,.\,C_l]\!]$ communicate on the private channel c_l. We have

$$\begin{aligned}
& [\![\mathsf{commit}\ V\ l\,(x').P_1 \mid P_2]\!]_a \mid [\![l\,.\,C_l]\!] \\
& \xrightarrow{\tau}\ \nu\, v_l\,.\,\nu\, w_l\,.\,(\varsigma(\mathsf{h}(s_l), \mathsf{h}(s_l + [\![V]\!]))_a \mid [\![P_1]\!]_a\,\{^{\mathsf{rd}(a\,,\,s_l\,,\,[\![V]\!]\,,\,w_l)}/_{x'}\}) \mid [\![P_2]\!]_a \\
& \mid (\{\mathsf{h}(a + \mathsf{h}(s_l))/l\} \mid \nu\, s\,.\,\{s/s_l\}) \\
=\ & \nu\, v_l.\nu\, w_l.[[\![P_1]\!]_a\{^{l\,.\,\mathbf{Rd}\,(a\ \ V)}/_{x'}\} \mid [\![P_2]\!]_a \mid l\,.\,Cap\,(a\ \ V)]\!] && \text{(B.2.19)} \\
=\ & \nu v_l.\nu w_l.[[\![P']\!]_a \mid l\,.\,C'_l]\!] && \text{(B.2.20)}
\end{aligned}$$

In (B.2.19), we apply structural equivalences to put the restrictions on v_l, w_l up to the top level, since they are fresh, and we fold the translation of the new committed location l. Indeed, our definitions state that $\varphi(M_1, M_2)_p = \{\mathsf{h}(p + M_1)/l\} \mid \varsigma(M_1, M_2)_p$. In (B.2.20), we apply the definitions of P'_a and $l.C'_l$.

Case communication

If A reduces by rule COMM for internal communication then for some $a_1, a_2, P_{11}, P_{12}, P_{21}, P_{22}$ we have $P_{a_1} = c!\langle M\rangle.P_{11} \mid P_{12}$ and $P_{a_2} = c?(x).P_{21} \mid P_{22}$ (a_1 and a_2 may be the same principal). After the communication, we have $P'_{a_1} = P_{11} \mid P_{12}$ and $P'_{a_2} = P_{21}\{^{M\sharp}/_{x}\} \mid P_{22}$. For all other principals a', $P'_{a'} = P_{a'}$. Since the communication is local, the sets of name $\mathcal{N}$ are $\mathcal{L}$ are the same.

We show that A can correctly perform a local communication since A can correctly output M, and then input M.

If the channel c is public ($c \notin \mathcal{N}$), the trace $\mathcal{C}_{a_1}[P_{a_1}] \xrightarrow{\nu\,\tilde{u}\,.\,c!\,M} \mathcal{C}'_{a_1}[P_{11} \mid P_{12}]$ is enabled for some well-formed context $\mathcal{C}'_{a_1}$. We can apply Lemma B.11: there is a trace $[\![\mathcal{C}_{a_1}]\!][[\![P]\!]_{a_1}] \xrightarrow{\psi}{}^{*} [\![\mathcal{C}'_{a_1}]\!][[\![P_{11} \mid P_{12}]\!]_{a_1}] = [\![\mathcal{C}'_{a_1}[P'_{a_1}]]\!]$.

We rewrite the resulting well-formed system as a context of a_2: Let $\mathcal{C}'_{a_2}$ be the source context for a_2 after the output:

$$\mathcal{C}'_{a_2}[\_] = \nu\mathcal{N}'\left(\textstyle\prod_{l\in\mathcal{L}'} l\,.\,C'_l \mid \prod_{a'\in\mathcal{A}\setminus\{a_1,a_2\}} a'[P'_{a'}] \mid a_1[P'_a 1] \mid a_2[\_] \mid \phi' \mid \mathcal{T}\right)$$

Since within $\mathcal{C}'_{a_1}$, for all principals a' other than a_1, $P'_{a'} = P_{a'}$, we have $[\![\mathcal{C}'_{a_1}[P'_{a_1}]]\!] \equiv \mathcal{C}'_{a_2}[P_{a_2}]$.

Now that process P_{a_1} sent M, principal a_2 is able to receive it, and does not learn anything new:the trace $\mathcal{C}'_{a_2}[c?(x).P_{21} \mid P_{22}] \xrightarrow{c?M} \mathcal{C}'_{a_2}[P_{21}\{^{M\sharp}/_{x}\} \mid P_{22}]$ is enabled, By Lemma B.11 again, there is a trace $[\![\mathcal{C}'_{a_2}[P_{a_2}]]\!] \xrightarrow{\psi}{}^{*} [\![\mathcal{C}'_{a_2}[P_{21}\{^{M\sharp}/_{x}\} \mid P_{22}]]\!] = \mathcal{C}'_{a_2}[P'_{a_2}] = A'$. Thus we have

$$A' = [\![\mathcal{C}_{a_1}[P_{a_1}]]\!] \xrightarrow{\psi}{}^{*} [\![\mathcal{C}'_{a_1}[P'_{a_1}]]\!] \equiv \mathcal{C}'_{a_2}[P_{a_2}] \xrightarrow{\psi'}{}^{*} [\![\mathcal{C}'_{a_2}[P'_{a_2}]]\!] = A'.$$

Case $\mathsf{if} - \mathsf{then} - \mathsf{else}$

Both conditional rules (IF_THEN and IF_ELSE) are simple homomorphisms according to our translation. By Lemma B.10, the equality is preserved by the translation of the source terms. Thus the translation of the conditional constructs behaves exactly as the source constructs.

Case resolution continuations Rule ADDRES allows to add a resolution continuation in the source system. The only condition on the added term H is that it must be known by the adversary (in particular it cannot contain secrets that belong to honest principals).

$$\frac{\mathsf{adversary\ knows}\ H}{0 \longrightarrow \mathsf{resolving}(H)} \quad \textsc{AddRes}$$

In the low level, this reduction is simulated by the resolution process receiving a message from the environment on channel log. Using the definition

$$R \stackrel{\text{def}}{=} (\mathsf{repl}\ log?(y_1).Q\,(\,y_1\,))\,|\,(\mathsf{repl}\ log!\langle\mathsf{None}\rangle)$$

we have

$$R \xrightarrow{log\,?\,(\,H\,)} R\,|\,Q\,(\,H\,)$$

Since every extended translation contains resolution, by STRUCT, we have that $[\![A]\!] \xrightarrow{log\,?\,(\,H\,)} [\![A]\!] \mid Q\,(\,H\,) \equiv [\![A \mid \mathsf{resolving}(H)]\!] \equiv [\![A']\!]$.

Rule DELRES allows to unconditionally discard a resolution continuation in the source system. System A is of the form $A \equiv A_0 \mid \mathsf{resolving}(H)$, for some A_0, H.

$$\frac{\mathsf{adversary\ knows}\ H}{0 \longrightarrow \mathsf{resolving}(H)} \quad \textsc{AddRes}$$

In the low level, we use the specially intended for this purpose replicated output $log!\langle\mathsf{ok}\rangle$ that communicates with the continuation.

$$R\,|\,Q\,(\,H\,) \xrightarrow{\tau} R\,|\,Q_1$$

where $Q1 =$ $\mathsf{if\ check\_idc}(H)\ \mathsf{and\ check\_idc}(\mathsf{ok})\ \mathsf{then}$
$\mathsf{if\ get\_idu}(H) = \mathsf{get\_idu}(\mathsf{ok})\ \mathsf{and}\ \mathsf{idc}_2(H) \neq \mathsf{idc}_2(\mathsf{ok})\ \mathsf{then}\ bad!\langle\mathsf{get\_prin}(H)\rangle$

Since ok is not an Idc, Q_1 reduces to empty process, and $R\,|\,Q\,(\,H\,) \xrightarrow{\tau}{}^{*} R$. Since every extended translation contains resolution, by STRUCT, we have that $[\![A_0 \mid \mathsf{resolving}(H)]\!] \xrightarrow{\tau}{}^{*} [\![A_0]\!] = [\![A']\!]$

□

PROOF OF THEOREM 3.1BIS Let A be a well-formed source system. Suppose $A \xrightarrow{\phi}{}^{*} A'$ for some trace ϕ and for some system A'. We proceed by induction on the length of the trace label ϕ to show that the translation of A can do all the transitions that A can do.

Base case If the length of ϕ is 1, Lemma B.12 or Lemma B.11 applies, if the transition is silent or not, respectively. So there exists a target level trace $[\![A]\!] \xrightarrow{\psi}{}^{*} [\![A']\!]$.

Inductive step We know that for all n, if the theorem holds for a source trace of length n then it also holds for traces of length $n+1$.

Suppose that for some n, we have $\phi = \phi_1.\phi_2$ where ϕ_1 is a sequence of n labels, and ϕ_2 is a label. According to the induction assumption, the theorem holds for $A \xrightarrow{\phi_1}{}^{*} A''$, and we have $[\![A]\!] \xrightarrow{\psi_1}{}^{*} [\![A'']\!]$. Since labelled transitions preserve well-formedness of source systems, system

A'' is still well-formed. We can apply Lemma B.11 (or Lemma B.12 for silent transitions) again on $A'' \xrightarrow{\phi_2} A'$ and obtain $[\![A'']\!] \xrightarrow{\psi_2}{}^{*} [\![A']\!]$. By composing the traces, we obtain $[\![A]\!] \xrightarrow{\psi_1}{}^{*} [\![A'']\!] \xrightarrow{\psi_2}{}^{*} [\![A']\!]$, that is $[\![A]\!] \xrightarrow{\psi}{}^{*} [\![A']\!]$ with $\psi = \psi_1.\psi_2$.

□

B.2.3 Security

Lemma B.13 (Security lemma)**.** *For all transition $[\![A]\!] \xrightarrow{\psi} S$ starting from a well-formed source system A, we have*

(1) *either there is a source transition $A \xrightarrow{\phi} A'$ leading to a well-formed source system A' such that $S \longrightarrow_D^* [\![A']\!]$;*

(2) *or $S \Downarrow e$ for some $e \notin \mathcal{A}$.*

Proof: Let A be a well-formed source system. Since A is well-formed, we have $A = \mathcal{C}_a[P_a]$ for a well formed source context $\mathcal{C}_a$ in which P_a is replaced with a hole:

$$\mathcal{C}_a[\_] = \nu\mathcal{N}.\left(\prod_{l\in\mathcal{L}} l.C_l \mid \prod_{a'\in\mathcal{A}\setminus\{a\}} a'[P_{a'}] \mid a[\_] \mid \phi \mid \mathcal{T}\right)$$

Suppose $[\![A]\!] \xrightarrow{\psi} S$ for some transition with label ψ. We perform a case analysis on ψ to find a matching source transition ϕ (or a target transition proving cheat).

Message input, using rule

$$\frac{}{c?(x).P \xrightarrow{c?(M)} P\{^{M}/_{x}\}} \text{ In}$$

By definition of the translation, the only input processes in evaluation context are those translated from the source system, using the same channel name, plus those introduced by the translation, that is, the replicated input on channel *log* in the resolution process R.

Consider the case where $\psi = c?(M)$ with $c \neq log$. Thus, for some a, P_1, P_2 we have $P_a = c?(x).P_1 \mid P_2$ which is translated to

$$[\![P]\!]_a \equiv \nu\, r\,.\,(c?(x).\mathsf{parse}\, x\, [\![P_1]\!]_a \mid \mathsf{repl}\, r?(\_).c?(x).\mathsf{parse}\, x\, [\![P_1]\!]_a) \mid [\![P_2]\!]_x.$$

After the input on c, the process $\mathsf{parse}\, M\, [\![P_1]\!]_a\, \{^{M}/_{x}\}$ runs, with different subcases depending on the received target term M. We recall the definition of parse:

$$\begin{array}{l}
\mathsf{parse}_1\, x\, P = \\
\quad \mathsf{if\ is_rd}(x) = \mathsf{ok\ then} \\
\quad\quad \mathsf{if\ check_idc}(\mathsf{get_idc}(x))\ \mathsf{then\ parse_c\ read}(x)\, P\ \mathsf{else}\ r!\langle\mathsf{None}\rangle \\
\quad \mathsf{else\ if\ is_idc}(x) = \mathsf{ok\ then\ if\ check_idc}(x)\ \mathsf{then}\ P\ \mathsf{else}\ r!\langle\mathsf{None}\rangle \\
\quad\quad \mathsf{else\ if\ is_pair}(x) = \mathsf{ok\ then\ parse}_1\ (+_1\, x)\ (\mathsf{parse}_1\ (+_2\, x)\ P) \\
\quad\quad\quad \mathsf{else\ parse_c}\, x\, P \\
\mathsf{parse}\, x\, P = \mathsf{parse}_1\, x\, (P \mid \mathsf{parse}_2\, x\,)
\end{array}$$

(1) **input Rd:** $M = \mathsf{rd}(M_1, M_2, M_3, M_4)$ for some terms M_1, M_2, M_3, and M_4. The first test is_rd succeeds.

If check_idc or $\mathsf{parse_c}$ fail, the translation emits on r and loops on sending. Otherwise we suppose that there is a source term M' such that $[\![M']\!] = M$.

Either M' is compatible with the context, and Lemma B.6 applies, or Lemma B.8 proves the detection of cheat by some external principal within M.

(2) **input Idc:** $M = \mathsf{idc}(M_1\,,\, M_2\,,\, M_3)$ for some terms M_1, M_2, and M_3. The first test fails, the second test succeeds, so after two reductions we obtain the process:

$$[\![P]\!]_a \xrightarrow{c\,?\,(M)} \xrightarrow{\tau} \nu\, r\,.\,(\mathsf{if}\ \mathsf{check\_idc}(M)\ \mathsf{then}\ [\![P_1\,\{^{M}/_{x}\}]\!]_a \mid \mathsf{parse}_2\,[\![M]\!]\ \mathsf{else}\ r!\langle\mathsf{None}\rangle \mid \mathsf{repl}\ r?(\_).c?(x).\mathsf{parse}\,x\,[\![P_1]\!]_a) \mid [\![P_2]\!]_a = P'.$$

The committed-capability integrity test is defined as follows:

$$\mathsf{check\_idc}(x) \stackrel{\text{def}}{=} \mathsf{verify}(\mathsf{idc}_2(x)\,,\, \mathsf{idc}_3(x)\,,\, \mathsf{idc}_1(x)) = \mathsf{ok}.$$

By instantiating x to $\mathsf{idc}(M_1\,,\, M_2\,,\, M_3)$, we obtain the test condition $\mathsf{verify}(M_2\,,\, M_3\,,\, M_1) = \mathsf{ok}$.

Discard If the condition equals false, the translation outputs on r and loops back to an input process on c.

$$P' \xrightarrow{\tau} \nu\, r\,.\,(r!\langle\mathsf{None}\rangle \mid \mathsf{repl}\ r?(\_).c?(x).\mathsf{parse}\,x\,[\![P_1]\!]_a) \mid [\![P_2]\!]_a \xrightarrow{\tau} [\![P]\!]_a$$

The input is thus discarded, and these transitions are simulated by doing no transition in the source system. In summary, we have

$$[\![P]\!]_a \xrightarrow{c\,?\,(M)} \xrightarrow{\tau_3} [\![P]\!]_a \text{ while } a[P_a] \rightarrow^{=} a[P_a]$$

Valid translation If the condition is satisfied, M_1 is the public key of some authenticated and auditable principal p, and for some l, V we received the translation of $l\,.\,\mathbf{Idc}\,(p\ M_2\ V) = M'$ (M_2 and V are empty if $p \in \mathcal{A}$). If the translation of A contains an active substitution $\{\mathsf{h}(M_1 + (+_1\, M_2))/l\}$, then we choose this l. Otherwise we choose l to be a fresh location name, so that $A \equiv \mathcal{C}_a[P_a] \mid l\,.\,\bot$ (by Fresh_loc).

We recall that $M = [\![M']\!] = M'^{\sharp}$.

– **Compatible** If $\mathcal{C}_a$ is compatible with M' then we can apply Lemma B.6 on compatible context updates.

$$[\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M'^{\sharp}]\!]\,[\![P_1]\!]_a\,\{^{[\![M'^{\sharp}]\!]}/_{x}\}] \xrightarrow{\tau}{}^{*} (\mathcal{C}_a \oplus M')[[\![P_1\,\{^{M'^{\sharp}}/_{x}\}]\!]_a]$$

Since r does not occur in $[\![P_1]\!]_a$, we can apply structural equivalences Dead_loop and Par.

$$\begin{aligned}&(\mathcal{C}_a \oplus M')[(\nu\, r\,.\,([\![P_1\,\{^{M'^{\sharp}}/_{x}\}]\!]_a \mid \mathsf{repl}\ r?(\_).c?(x).\mathsf{parse}\,x\,[\![P_1]\!]_a)) \mid [\![P_2]\!]_a]\\ &\equiv (\mathcal{C}_a \oplus M')[[\![P_1\,\{^{M'^{\sharp}}/_{x}\}]\!]_a \mid [\![P_2]\!]_a] = A'\end{aligned}$$

We can build the input source transition : $\mathcal{C}_a[P_a] \xrightarrow{c?M'} (\mathcal{C}_a \oplus M')[P_1\{^{M'^{\sharp}}/_{x}\} \mid P_2]$, by applying Receive_term, and then for all $l \in \mathcal{L}$, Up_in in conjunction with either In_owned or In.

– **Cheating** otherwise, $C_l \curlyvee \mathsf{locs}(l, M')$ does not exist, and $p = \mathsf{prin\_of}(C_l) \notin \mathcal{A}$. By construction (choice of l), these capabilities are related (Def B.2.3). Lemma B.8 applies, and the owner is blamed:

$$[\![\mathcal{C}_a]\!][\mathsf{parse}\,[\![M']\!]\,[\![P_1\,\{^{M'}/_{x}\}]\!]_a] \Downarrow \mathsf{get\_prin}(l\,.\,C_l)$$

Thus we also have $[\![A]\!] \Downarrow \mathsf{get\_prin}(l\,.\,C_l)$.

(3) **input prin:** principal $M = \mathsf{prin}(M_1)$

The first three tests fail, and parse reduces to $\mathsf{parse_c}$.

$$\mathsf{parse}\,M\,[\![P_1]\!]_a\,\{^{M}/_{x}\} \xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} \mathsf{parse_c}\,M\,(P_1\,\{^{M}/_{x}\} \mid \mathsf{parse}_2\,M\,).$$

Within $\mathsf{parse_c}$, the first test fails, the second one ($\mathsf{is\_prin}$) succeeds.
If the $\mathsf{is\_pk}(M)$ test succeeds, $[\![A]\!]$ contains an active substitution $\{\mathsf{prin}(M_1)/p\}$ for some p, and p is a known principal within A, with $[\![p]\!] = p$. Since $\forall l \in \mathcal{L}, \mathsf{locs}(l, p) = \bot$, Lemma B.4 applies: $[\![\mathcal{C}_a]\!][\mathsf{parse_c}\ M\ (P_1\ \{^M/_x\} \mid \mathsf{parse}_2\ M\ )] \xrightarrow{\tau}{}^* [\![\mathcal{C}_a]\!][P_1\ \{^M/_x\} \mid \mathsf{parse}_2\ M\ ] \xrightarrow{\tau}{}^* [\![\mathcal{C}_a]\!][P_1\ \{^M/_x\}]$. We can get rid of the replicated input process by DEAD_LOOP. The matching source transition exists: $\mathcal{C}_a[P_a] \xrightarrow{c?p} \mathcal{C}_a[P_1\{^p/_x\} \mid P_2]$.
If the $\mathsf{is\_pk}(M)$ test succeeds, the principal represented by M is unknown within $\mathcal{C}_a$. The test $\mathsf{is\_pair}(M)$ necessarily fails, so there is an output on r and the translation loops. In summary, $[\![P]\!]_a \xrightarrow{c\,?\,(\,M\,)} \xrightarrow{\tau_6} [\![P]\!]_a$ while $\mathcal{C}_a[P_a] \xrightarrow{c?p} \mathcal{C}_a[P_a]$.

(4) **input Idu:** $M = \mathsf{idu}(M_1)$ for some M_1. Semantically we can distinguish two cases: M_1 is a valid hash of some principal's name concatenated with its target secret seed, or it is not. We apply the following reasoning for both cases (but in the second case the corresponding source location cannot ever be committed).
For some l, we consider a source label $M' = l\,.\,\mathbf{Idu}$, or $M' = l\,.\,\mathbf{Idu}\,(e\ M_1)$ if owned by some external principal e. If the translation of A contains an active substitution $\{M_1/l\}$, then we choose this l, otherwise we choose a fresh l. Committable terms are compatible with any context, so Lemma B.6 applies and yields

$$[\![\mathcal{C}_a]\!][\mathsf{parse}\ [\![M'^\sharp]\!]\ [\![P_1\ \{^{M'^\sharp}/_x\}]\!]_a] \xrightarrow{\tau}{}^* (\mathcal{C}_a \oplus M')[[\![P_1\ \{^{M'^\sharp}/_x\}]\!]_a]$$

We can get rid of the replicated input process by DEAD_LOOP, so that $[\![\mathcal{C}_a]\!][[\![P]\!]_a] \xrightarrow{c\,?\,(\,\mathsf{idu}(M_1)\,)}{}^*$ $(\mathcal{C}_a \oplus M')[[\![P_1\ \{^{M'^\sharp}/_x\} \mid P_2]\!]_a]$ and $A \xrightarrow{c?M'} (\mathcal{C}_a \oplus M')[P_1\{^{M'^\sharp}/_x\} \mid P_2]$.

(5) **input pair:** $M = M_1 + M_2$
The first three tests fail, the fourth ($\mathsf{is\_pair}$) succeeds.
$\mathsf{parse}\ M_1 + M_2\ [\![P_1]\!]_a\ \{^M/_x\} = (\mathsf{parse}_1\ M_1\ \mathsf{parse}_1\ M_2\ [\![P_1]\!]_a\ \{^M/_x\}) \mid \mathsf{parse}_2\ M_1 + M_2$
If $[\![A]\!] \xrightarrow{c\,?\,(\,M_1\,)} S_1$ then
- M1 is discarded
- M1 is valid: M1 is compatible (M2 is discarded, or M2 is valid and (compatible : M1 + M2 or Loop)) or Loop

(6) **other input:** all tests in $\mathsf{parse_1}$ and $\mathsf{parse_c}$ fail, the translation loops.
$[\![P]\!]_a \xrightarrow{c\,?\,(\,M\,)} \xrightarrow{\tau_6} [\![P]\!]_a$ while $\mathcal{C}_a[P_a] \xrightarrow{c?p} \mathcal{C}_a[P_a]$.

We now consider the case when $c = log$. Receiving on log only occurs within the resolution process. There are two cases: the reception is done either by an entire copy of resolution process, or by one of its continuations.

Suppose that there is a committed location $l \in \mathcal{L}$, and C_l is committed ($C_l = \mathbf{Idc}\,(p\ H\ V)$ or $C_l = \mathbf{Rd}\,(p\ H\ V)$).

– If the input is done by a copy of R, we have

$$R = \mathsf{repl}\ log?(x).Q\ (\,x\,) \xrightarrow{log\,?\,(\,M\,)} R \mid Q\ (\,M\,)$$

Since M can be received by the translation, by rule ADDRES we can build the source transition $A \xrightarrow{\tau} A \mid \mathsf{resolving}(M)$.

– We recall the definition of Q:

$$Q\ (\,y_1\,) \stackrel{\text{def}}{=} \quad log?(y_2).\mathsf{if}\ \mathsf{check\_idc}(y_1)\ \mathsf{and}\ \mathsf{check\_idc}(y_2)\ \mathsf{then}$$
$$\mathsf{if}\ \mathsf{get\_idu}(y_1)\ =\ \mathsf{get\_idu}(y_2)\ \mathsf{and}\ \mathsf{idc}_2(y_1) \neq \mathsf{idc}_2(y_2)\ \mathsf{then}\ bad!\langle\mathsf{get\_prin}(y_1)\rangle$$

Thus if the input is done by a continuation $Q\,(\,M'\,)$, we instantiate y_1 to M', and y_2 to M in all the tests. If M and M' are valid translations of some source Idcs that refer to the same location owned by principal e and are committed to different values, then $Q\,(\,M'\,) \Downarrow e$. Otherwise, some of the resolution tests fail. We build a source transition $A \xrightarrow{\tau} A'$ where A is a well-formed system whose components are all the same as A except from $\mathcal{T}' = \mathcal{T} \setminus \mathsf{resolving}(M')$.

Silent transition A silent transition $\psi = \tau$ is due to an internal reduction within the translation of A. By definition of the translation, the only reductions possible within a translated source system are by the following target rules.

(1) Comm

Internal communications happen while committing a location, or –transparently– while auditing the system by the resolution process.

Commitment of a location For some a, P_1, P_2, l, V we have $P_a = \mathsf{commit}\ V\ l\,(x).P_1 \mid P_2$ which is translated (using Lemma B.3) to

$$[\![P]\!]_a = (c_l?(\_).\nu\, v_l \,.\, \nu\, w_l \,.\, (\varsigma(\mathsf{h}(s_l), \mathsf{h}(s_l + [\![V]\!]))_a \mid [\![P_1]\!]_a \,\{^{\mathsf{rd}(a\,,\,s_l\,,\,[\![V]\!]\,,\,w_l)}/_x\})) \mid [\![P_2]\!]_a$$

Since $l \in \mathcal{L}$, $C_l = Cap\,(a\ \ V)$ exists; V may be empty.

– $C_l = Cap\,(a\,)$: l has not been committed yet, $Cap \in \{0, \mathbf{Idu}\}$. We have

$$[\![l \,.\, Cap\,(a\,)]\!] = c_l!\langle \mathsf{None}\rangle \mid \{\mathsf{h}(a + \mathsf{h}(s_l))/l\} \mid \nu\, s \,.\, \{s/s_l\}$$

The location l and the committing subprocess can communicate on c_l:

$$\begin{aligned}[\![l \,.\, Cap\,(a\,)]\!] | [\![P]\!]_a &\xrightarrow{\tau} \{\mathsf{h}(a + \mathsf{h}(s_l))/l\} \mid \nu\, s \,.\, \{s/s_l\} \\ &\quad \mid \nu\, v_l \,.\, \nu\, w_l \,.\, (\varsigma(\mathsf{h}(s_l), \mathsf{h}(s_l + [\![V]\!]))_a \mid [\![P_1]\!]_a \,\{^{\mathsf{rd}(a\,,\,s_l\,,\,[\![V]\!]\,,\,w_l)}/_x\}) \mid [\![P_2]\!]_a \\ \equiv \nu\, v_l \,.\, \nu\, w_l \,.\, ((\varphi(\mathsf{h}(s_l), \mathsf{h}(s_l + [\![V]\!]))_a \mid \nu\, s \,.\, \{s/s_l\} \mid [\![P_1 \,\{^{\mathsf{rd}(a\,,\,s_l\,,\,[\![V]\!]\,,\,w_l)}/_x\}]\!]_a) \mid [\![P_2]\!]_a) \\ \equiv [\![l \,.\, Cap\,(a\ \ V)]\!] \mid [\![P_1\{^{l\,.\,\mathbf{Rd}\,(a\ \ V)}/_x\} \mid P_2]\!]_a\end{aligned}$$

Commitment updates the location l in A, therefore we have:

$$A \xrightarrow{\tau} \nu\mathcal{N}\left(\textstyle\prod_{l' \in \mathcal{L}\setminus\{l\}} l \,.\, C_l | \prod_{a' \in \mathcal{A}\setminus\{a\}} a'[P_{a'}] \| l \,.\, Cap\,(a\ \ V) \| a[P_1\{^{l\,.\,\mathbf{Rd}\,(a\ \ V)}/_x\} \mid P_2] | \phi \mid \mathcal{T}\right).$$

– $C_l = Cap\,(a\ \ V)$: l has already been committed, for all $Cap \neq \bot$. We have

$$[\![l \,.\, Cap\,(a\ \ V)]\!] = \varphi(\mathsf{h}(s_l), \mathsf{h}(s_l + [\![V]\!]))_a \mid \nu\, s \,.\, \{s/s_l\}$$

The translation of the committed location l does not send anything on c_l, so the translation of the committing process is blocked. The source system trying to commit a committed location is also stuck: the behaviour is the same.

resolution Within our translation, committed locations emit repeatedly their Idcs on *log* and the resolution process, or its continuations read Idcs from *log*. There are two cases: the silent transition results from communication of a committed location with either an entire copy of resolution process, or with one of its continuations.

Suppose that there is a committed location $l \in \mathcal{L}$. We have either $C_l = \mathbf{Idc}\,(p\ H\ V)$, or $C_l = \mathbf{Rd}\,(p\ H\ V)$, whose translations both include φ that provides valid active substitutions for l, v_l, and w_l, and also makes replicated outputs the **idc** of the capability : $\mathsf{idc}(p\,,\ v_l\,,\ w_l)$ on channel *log*.

– If location l communicates with a copy of R, we have

$$[\![l\,.\,C_l]\!] \mid R = [\![l\,.\,C_l]\!] \mid \mid log?(x).Q\,(\,x\,) \xrightarrow{\tau} [\![l\,.\,C_l]\!] | Q\,(\,\mathsf{idc}(p\,,\,v_l\,,\,w_l)\,)$$

We build the source transition $A \xrightarrow{\tau} A \mid \mathsf{low}(Q\,(\,\mathsf{idc}(p\,,\,v_l\,,\,w_l)\,))$.

– Suppose that location l communicates with a continuation of the resolution process: $[\![l\,.\,C_l]\!] | Q\,(\,M\,)$. If M is a translation of some source capability $l.C$ such that $C_l \curlyvee C$ does not exist, then Lemma B.7 applies, and a principal is accused:

$$Q\,(\,[\![l\,.\,\mathcal{C}]\!]\,) | [\![l\,.\,C_l]\!] \Downarrow \mathsf{prin\_of}(C)$$

Otherwise, the continuation is consumed and erased from the source system:

$$Q\,(\,[\![l\,.\,\mathcal{C}]\!]\,) | [\![l\,.\,C_l]\!] \xrightarrow{\tau} *[\![l\,.\,C_l]\!]$$

Then we build a source transition $A \xrightarrow{\tau} A'$ where $\mathcal{T}' = \mathcal{T} \setminus \mathsf{resolving}([\![l\,.\,C]\!])$.

(2) If_then

Positive branch of a conditional construct in the translation is taken in two cases:

– for some $a, P_1, P_2, P_3, M_1, M_2$ there is $P_a = \mathsf{if}\ M_1 = M_2\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \mid P_3$ and the equational target theory justifies $[\![M_1]\!] = [\![M_2]\!]$.
The translation of P_a makes a silent step to continue with the first branch: $\mathsf{if}\ [\![M_1]\!] = [\![M_2]\!]\ \mathsf{then}\ [\![P_1]\!]_a\ \mathsf{else}\ [\![P_2]\!]_a \xrightarrow{\tau} [\![P_1]\!]_a$. By Lemma B.10, $[\![M_1]\!] = [\![M_2]\!]$ if and only if $M_1 = M_2$, thus in the source system, we have the matching transition $a[\mathsf{if}\ M_1 = M_2\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \mid P_3] \xrightarrow{\tau} a[P_1 \mid P_3]$, and so $A \xrightarrow{\tau} \mathcal{C}_a[P_1 \mid P_3]$.

– for some a, P_1, P_2 there is $P_a = \mathsf{newloc}\,(x, y).P_1 \mid P_2$, translated with

$$[\![\mathsf{newloc}\,(x, y).P_1]\!]_a = \nu\, s_{l_0'}\,.\,\nu\, c_{l_0}\,.\,\tau.c_{l_0}!\langle\mathsf{None}\rangle \mid [\![P_1]\!]_a\,\{^{c_{l_0}}/_{c_x}\}\,\{^{s_{l_0'}}/_{s_x}\}\,\{^{\mathsf{h}(a+\mathsf{h}(s_{l_0'}))}/_{l_0}\}\,\{^{\mathsf{idu}(l_0)}/_{y}\}$$

As $\tau.A$ abbreviates $\mathsf{if}\ n = n\ \mathsf{then}\ A$ which can only reduce into A, the rule If_then always applies. By Lemma B.2, we have

$$[\![\mathsf{newloc}\,(x, y).P]\!]_a \longrightarrow \nu\, s_{l_0}\,.\,\nu\, c_{l_0}\,.\,\nu\, l_0\,.\,([\![a[P\{^{l_0}/_x\}\{^{l_0\,.\,\mathbf{Idu}}/_y\}]\!] \mid [\![l_0\,.\,0\,(a\,)]\!])$$

This silent transition updates the context with a fresh location, precisely

$$A \xrightarrow{\tau} \nu\mathcal{N} \cup \{l_0\}\left(\prod_{l\in\mathcal{L}} l\,.\,C_l | \prod_{a'\in\mathcal{A}\setminus\{a\}} a'[P_{a'}] | l_0\,.\,0\,(a\,) | a[P_1\{^{l_0\,.\,\mathbf{Idu}}/_x\} \mid P_2] | \phi \mid \mathcal{T}\right).$$

(3) If_else

Negative branch of a conditional construct in the translation corresponds to a negative conditional in the source system.

For some $a, P_1, P_2, P_3, M_1, M_2$ there is $P_a = \mathsf{if}\ M_1 = M_2\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \mid P_3$ and the equational target theory justifies $[\![M_1]\!] \neq [\![M_2]\!]$.

The translation of P_a makes a silent step to continue with the second branch: $\mathsf{if}\ [\![M_1]\!] = [\![M_2]\!]\ \mathsf{then}\ [\![P_1]\!]_a\ \mathsf{else}\ [\![P_2]\!]_a \xrightarrow{\tau} [\![P_2]\!]_a$. Since our translation of terms is secure, that is if $[\![M_1]\!] \neq [\![M_2]\!]$ then $M_1 \neq M_2$, in the source system, we have the matching transition $a[\mathsf{if}\ M_1 = M_2\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \mid P_3] \xrightarrow{\tau} a[P_2 \mid P_3]$, so $A \xrightarrow{\tau} \mathcal{C}_a[P_2 \mid P_3]$.

Output transition using rule

$$\frac{}{c!\langle M\rangle.P \xrightarrow{c!<M>} P}\ \textsc{Out}$$

By definition of the translation, the only output processes in evaluation context are those translated from the source system, using the same channel name, plus those introduced by the translation, that is, outputs on channel *log* in the resolution process R.

Consider the case where $\psi = c!M$ with $c \neq log$ and $c \neq bad$. Thus, for some a, P_1, P_2 we have $P_a = c!\langle M\rangle.P_1 \mid P_2$ which is translated to

$$[\![P]\!]_a \equiv c!\langle M\rangle.[\![P_1]\!]_a \mid [\![P_2]\!]_x.$$

for some $c, \tilde{u}, a, P_1, P_2$ and for some M' such that $M = [\![M']\!]$. According to Lemma B.9, there is a well-formed source context $\mathcal{C}'_a$ such that $[\![\mathcal{C}_a]\!][c!\langle M\rangle.[\![P_1]\!]_a] \xrightarrow{\nu\tilde{u}.\, c!<M>} [\![\mathcal{C}'_a]\!][[\![P_1]\!]_a]$. The corresponding source transition is $\mathcal{C}_a[P_1] \xrightarrow{\nu\tilde{u}.\, c!\, M'} \mathcal{C}'_a[P_1]$, so we also have $A \xrightarrow{\nu\tilde{u}.\, c!\, M'} \mathcal{C}'_a[P_1 \mid P_2] \equiv A'$.

We now consider the case of an output on channel *log*. If $[\![A]\!] \xrightarrow{log!M} S$, then it comes from the translation of some committed location $l.Cap(pM'V)$ such that $M = [\![l\,.\,\mathbf{Idc}\,(p\,M'')]\!]$ and $\mathbf{Idc}\,(p\,M'') \preceq Cap\,(p\,M'\,V)$. All emissions on *log* are replicated, so we have $S = [\![A]\!]$ by the rule $\mathsf{repl}\,P \equiv P \mid \mathsf{repl}\,P$, and thus $A \rightarrow^{=} A$.

Note that the translation of a source process can only output on *bad* after a series of silent transitions.

□

Proof of Theorem 3.2bis Let A be a well-formed source system. Suppose $[\![A]\!] \xrightarrow{\psi} S$ for some series of transitions with labels ψ. We proceed by induction on the length of the label.

Base case If the length of ψ is 1, Lemma B.13 applies.

Inductive step We know that for all n, if the theorem holds for a source trace of length n then it also holds for traces of length $n+1$.

Suppose that for some n, we have $\psi = \psi_1.\psi_2$ where ψ_1 is a sequence of n labels, and ψ_2 is a label.

According to the induction assumption, the theorem holds for $[\![A]\!] \xrightarrow{\psi_1}{}^{*} S$, and we have

- $A \xrightarrow{\phi_1}{}^{*} A''$ for some well-formed A'' such that $S \longrightarrow_D^* [\![A'']\!]$.
  There are two cases.
  - If $S = [\![A'']\!]$, then we can apply Lemma B.13 to $[\![A'']\!] \xrightarrow{\psi_1} S'$ as A'' is still a well-formed source system. Again, there are two cases: either there is a corresponding source transition $A'' \xrightarrow{\phi_2}{}^{*} A'$ for some well-formed A' such that $S' \longrightarrow_D^* [\![A']\!]$. Thus $A \xrightarrow{\phi_1}{}^{*} \xrightarrow{\phi_2} A'$.
    Or there is a cheating principal blamed: $S' \Downarrow e$ for some $e \notin \mathcal{A}$.
  - Otherwise, S is not a translation of a source system, then $S \xrightarrow{\psi_2} S'$ is a deterministic transition going from S and thus S' has the same behaviour as S', so the theorem holds.
- or $S \Downarrow e$ for some $e \notin \mathcal{A}$. Since logging evidence and the resolution process is replicated, blaming e is still enabled after the transition ψ_1. So the theorem also holds for $[\![A]\!] \xrightarrow{\psi} S'$.

□

Proof of Theorem 3.1(Functional adequacy) Let A be a well-formed source system as defined in Section 3.2.5. Let $A \xrightarrow{\phi}{}^{*} A'$ be a series of source transitions. Since A and A' can be seen as an extended source system with an empty set of resolution continuations ($\mathcal{T} = \varnothing$), and since traces of the extended systems include ϕ^*, Theorem 3.1bis applies. We have that $[\![A]\!] \xrightarrow{\phi}{}^{*} [\![A']\!]$.

□

Theorem B.14 (Weakened Theorem 3.2)**.** *For all transitions $[\![A]\!] \xrightarrow{\psi}{}^{*} S$ starting from a well-formed source system A, we have*

(1) *either there are source transitions* $A \xrightarrow{\phi}{}^{*} A'$ *leading to a well-formed source system* A' *such that for some resolution store* $\mathcal{T}$ $S \longrightarrow_D^* [\![A' \mid \mathcal{T}]\!] \xrightarrow{\tau}{}^{*} [\![A']\!]$*; or* $S \Downarrow e$ *for some* $e \notin \mathcal{A}$*;*

(2) *if* $S \Downarrow M$*, then* $M \notin \mathcal{A}$*.*

PROOF: Let A be a well-formed source system as defined in Section 3.2.5. Let $[\![A]\!] \xrightarrow{\psi}{}^{*} S$ be low-level transitions. Since A can be seen as an extended source system with an empty set of resolution continuations, Theorem 3.2bis applies. We have

(1) either there are source transitions $A \xrightarrow{\phi}{}^{*} A' \mid \mathcal{T}$ leading to a well-formed extended source system A' such that $S \longrightarrow_D^* [\![A' \mid \mathcal{T}]\!]$; or $S \Downarrow e$ for some $e \notin \mathcal{A}$;
In the latter case, the proof is finished. In the former case, we can discard the resolution continuations in $\mathcal{T}$ using Lemma B.15: $A' \mid \mathcal{T} \xrightarrow{\phi'}{}^{*} A'$.By Theorem 3.1 we have that $[\![A' \mid \mathcal{T}]\!] \xrightarrow{\tau}{}^{*} [\![A']\!]$. By transitivity, we have $A \xrightarrow{\phi'}{}^{*} A'$.

(2) if $S \Downarrow M$, then $M \notin \mathcal{A}$.

□

Lemma B.15 (Discarding transitions related to low-level records)**.** *For all well-formed source systems* A *and* A' *that contain no target continuations, let* $\mathcal{T}$ *be a non-empty parallel composition of target continuations, we have that if there is a series of labelled transitions* $A \xrightarrow{\phi} A' \mid \mathcal{T}$ *then there is a series of transitions* $A \xrightarrow{\phi'} A'$*.*

PROOF: Let A and A' be well-formed source systems that contain no target continuations, let $\mathcal{T}$ be a set of target continuations, and $A \xrightarrow{\phi} A' \mid \mathcal{T}$. We proceed by induction on the length of ϕ. If ϕ is empty, then $A' = A$, $\mathcal{T} = 0$.

Base case. If the length of ϕ is 1, then necessarily the used transition rule is ADDRES. Since we have $A' = A$ and $\phi = \tau$, me also have $A \xrightarrow{\epsilon} A'$.

Inductive case. Suppose that the lemma is true when the length of the label is n. Suppose that $A \xrightarrow{\phi}\xrightarrow{\phi_1} A' \mid \mathcal{T}$ when the length of ϕ is n and the length of ϕ_1 is 1. For some well-formed A_1 with any target continuations and for some $\mathcal{T}_1$ we have $A \xrightarrow{\phi} A_1 \mid \mathcal{T}_1$. By the induction hypothesis, there is a series of transitions $A \xrightarrow{\phi'_1} A_1$. We examine the rule used for the transition $A_1 \mid \mathcal{T}_1 \xrightarrow{\phi_1} A' \mid \mathcal{T}$.

If rules ADDRES or DELRES are used, then $A' = A1$, and we have $A \xrightarrow{\phi'_1} A'$.

If any other rule is used, then $\mathcal{T} = \mathcal{T}_1$, and so by PAR we have $A_1 \xrightarrow{\phi_1} A'$. By composing with $A \xrightarrow{\phi'_1} A_1$ we obtain $A \xrightarrow{\phi'_1}\xrightarrow{\phi_1} A'$. □

Theorem B.16 (Weakened security 2)**.** *For all transitions* $[\![A]\!] \xrightarrow{\psi}{}^{*} S$ *starting from a well-formed source system* A*, we have*

(1) *either there are source transitions* $A \xrightarrow{\phi}{}^{*} A'$ *leading to a well-formed source system* A' *such that* $S \longrightarrow_D^* [\![A' \mid \mathcal{T}]\!]$ *and* $[\![A']\!] \xrightarrow{\psi'}{}^{*} [\![A' \mid \mathcal{T}]\!]$*; or* $S \Downarrow e$ *for some* $e \notin \mathcal{A}$*;*

(2) *if* $S \Downarrow M$*, then* $M \notin \mathcal{A}$*.*

PROOF: Let A be a well-formed source system as defined in Section 3.2.5. Let $[\![A]\!] \xrightarrow{\psi}{}^{*} S$ be low-level transitions. Since A can be seen as an extended source system with an empty set of resolution continuations, Theorem B.14 applies. We have

(1) either there are source transitions $A \xrightarrow{\phi}{}^{*} A' \mid \mathcal{T}$ leading to a well-formed extended source system $A' \mid \mathcal{T}$ such that $S \longrightarrow_D^* [\![A' \mid \mathcal{T}]\!]$; or $S \Downarrow e$ for some $e \notin \mathcal{A}$;
In the latter case, the proof is finished. In the former case, we can discard the resolution continuations using Lemma B.15: $A \xrightarrow{\tau}{}^{*} A'$, and also $A' \xrightarrow{\tau}{}^{*} A' \mid \mathcal{T}$.By Theorem 3.1 we have that $[\![A']\!] \xrightarrow{\psi'}{}^{*} [\![A' \mid \mathcal{T}]\!]$.

(2) if $S \Downarrow M$, then $M \notin \mathcal{A}$.

□

Appendix C

Offline e-cash

C.1 Semantics of source language

$\boxed{A \rightarrow_a A'}$ A reduces to A'

$$\frac{(\, n \notin fn(P_1) \,\wedge\, n \notin fn(P_2)\,)}{\mathcal{U}[\mathsf{withdraw!}\,\mathcal{B}\,(x)\,.P_1] \mid \mathcal{B}[\mathsf{withdraw?}\,\mathcal{U}.P_2] \rightarrow_a \nu\, l\,.\,(\mathcal{U}[P_1\{^{l}/_{x}\}] \mid \mathcal{B}[P_2] \mid l.(\mathcal{B}\,\mathcal{U}\,\varnothing))}\ \textsc{W}$$

$$\frac{(\, s \notin fn(P)\,)}{\mathcal{U}[\mathsf{spend!}\,\mathcal{B}\,\mathcal{M}\,c\,l] \mid \mathcal{M}[\mathsf{spend?}\,\mathcal{B}\,(x)\,(x_1)(x_2).P] \mid l.(\mathcal{B}\,\mathcal{U}\,\varnothing) \rightarrow_a \nu\, s\,.\,(\mathcal{M}[P\{^{c}/_{x}\}\{^{\langle l,s\rangle}/_{x_1}\}\{^{\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)}/_{x_2}\}] \mid l.(\mathcal{B}\,\mathcal{U}\,\{\,s\,\}))}\ \textsc{S}$$

$$\frac{}{\mathcal{M}[\mathsf{deposit!}\,\mathcal{B}\,\langle l,s\rangle] \mid \mathcal{B}[\mathsf{deposit?}\,\mathcal{M}\,(x).P] \mid l.(\mathcal{B}\,\mathcal{U}\,\{\,s\,\}) \rightarrow_a \mathcal{B}[P\{^{\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)}/_{x}\}] \mid l.(\mathcal{B}\,\mathcal{U}\,\{\,\overline{s}\,\})}\ \textsc{D}$$

$$\frac{}{p_1[c\,!\,M] \mid p_2[c?(x).P] \rightarrow_a p_2[P\{^{M}/_{x}\}]}\ \textsc{Comm}$$

$$\frac{}{p[\mathsf{if}\ M = M\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2] \rightarrow_a p[P_1]}\ \textsc{If\_then}$$

$$\frac{\neg\, M_1 = M_2}{p[\mathsf{if}\ M_1 = M_2\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2] \rightarrow_a p[P_2]}\ \textsc{If\_else}$$

$\boxed{A \xrightarrow{\phi}_a A'\ \mathit{condition}}$ A makes a ϕ transition to A'

$$\frac{A \rightarrow_a A'}{A \xrightarrow{\tau}_a A'}\ \textsc{AltSilent}$$

$$\frac{}{\mathcal{U}[\mathsf{withdraw!}\,\mathcal{B}\,(x)\,.P] \xrightarrow{\mathsf{withdraw!}\,\mathcal{U}\,\mathcal{B}}_a \nu\, l\,.\,(\mathcal{U}[P\{^{l}/_{x}\}] \mid l.(\mathcal{B}\,\mathcal{U}\,\varnothing))\,(\,l \notin fn(P)\,)}\ \textsc{AltWithdraw}$$

$$\frac{}{\mathcal{B}[\mathsf{withdraw?}\,\mathcal{U}.P] \xrightarrow{\nu l.\mathsf{withdraw?}\,\mathcal{B}\,\mathcal{U}}_a \mathcal{B}[P] \mid l.(\mathcal{B}\,\mathcal{U}\,\varnothing)}\ \textsc{AltWithdrawn}$$

$$\frac{}{\mathcal{U}[\mathsf{spend!}\,\mathcal{B}\,\mathcal{M}\,c\,l] \mid l.(\mathcal{B}\,\mathcal{U}\,\varnothing) \xrightarrow{\nu s.\mathsf{spend!}\,\mathcal{M}\,\mathcal{B}\,l\,s}_a l.(\mathcal{B}\,\mathcal{U}\,\{\,s\,\})}\ \textsc{AltSpendSimple}$$

$$\frac{}{\mathcal{U}[\mathsf{spend!}\,\mathcal{B}\,\mathcal{M}\,c\,l] \mid l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}) \xrightarrow{\nu s.\mathsf{spend!}\,\mathcal{M}\,\mathcal{B}\,l\,s}_a l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}\uplus\{\,s\,\})}\ \textsc{AltSpend}$$

$$\frac{}{\mathcal{M}[\mathsf{spend?}\,\mathcal{B}\,(x)\,(x_1)(x_2).P] \mid l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}) \xrightarrow{\nu s.\mathsf{spend?}\,\mathcal{M}\,\mathcal{B}\,c\,l\,s\,\mathcal{U}}_a \mathcal{M}[P\{^{c}/_{x}\}\{^{\langle l,s\rangle}/_{x_1}\}\{^{\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)}/_{x_2}\}] \mid (l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}\uplus\{\,s\,\}))}\ \textsc{AltSpent}$$

$$\frac{}{\mathcal{M}[\mathsf{deposit!}\,\mathcal{B}\,\langle l,s\rangle] \mid l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}) \xrightarrow{\mathsf{deposit!}\,\mathcal{M}\,\mathcal{B}\,l\,s}_a l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}\uplus\{\,\overline{s}\,\})}\ \textsc{AltDeposit}$$

$$\frac{}{\mathcal{M}[\mathsf{deposit!}\,\mathcal{B}\,\langle l,s\rangle] \mid l.(\mathcal{B}\,\mathcal{U}\,\{\,s\,\}) \xrightarrow{\mathsf{deposit!}\,\mathcal{M}\,\mathcal{B}\,l\,s}_a l.(\mathcal{B}\,\mathcal{U}\,\{\,\overline{s}\,\})}\ \textsc{AltDepositSimple}$$

$$\frac{}{\mathcal{B}[\mathsf{deposit?}\,\mathcal{M}\,(x).P] \mid l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}) \xrightarrow{\mathsf{deposit?}\,\mathcal{B}\,\mathcal{M}\,l\,s\,\mathcal{U}}_a \mathcal{B}[P\{^{\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)}/_{x}\}] \mid l.(\mathcal{B}\,\mathcal{U}\,\mathcal{I}\uplus\{\,\overline{s}\,\})}\ \textsc{AltDepositedD}$$

$$\frac{}{\mathcal{A}[c\,!\,M] \xrightarrow{c\,!\,M}_a \mathcal{A}[0]}\ \textsc{AltSend\_term}$$

$$\frac{A \xrightarrow{c\,!\,M}_a A' \wedge (\, r \in fnfv(M) \wedge c \neq r\,)}{\nu\, r\,.\,A \xrightarrow{\nu r.c\,!\,M}_a A'}\ \textsc{AltOpen}$$

$$\frac{}{\mathcal{A}[c?(x).P] \xrightarrow{c\,?\,p}_a \mathcal{A}[P\{^{p}/_{x}\}]}\ \textsc{AltReceive\_term}$$

$$\frac{}{\mathcal{A}[c?(x).P] \mid l.C_l \xrightarrow{c\,?\,\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)}_a \mathcal{A}[P\{^{\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)}/_x\}] \mid (l.C_l \curlyvee l.(\mathcal{B}\,\mathcal{U}\,\{\,s\,\}))} \text{ ALTRECV\_KNOWN\_SPT}$$

$$\frac{A \xrightarrow{\phi}_a A' \wedge c \notin \phi}{\nu\, c\,.\, A \xrightarrow{\phi}_a \nu\, c\,.\, A'} \text{ ALTSCOPE}$$

$$\frac{A_1 \xrightarrow{\phi}_a A_1' \wedge (bvbn(\phi) \cap bvbn(A_2)) = \varnothing}{A_1 \mid A_2 \xrightarrow{\phi}_a A_1' \mid A_2} \text{ ALTPAR}$$

$$\frac{A_1 \equiv A_2 \wedge (\, A_2 \xrightarrow{\phi}_a A_3 \wedge A_3 \equiv A_4\,)}{A_1 \xrightarrow{\phi}_a A_4} \text{ ALTLAB\_STRUCT}$$

C.2 Detectability

Lemma C.1 (Functional adequacy)**.** *Le A be a high level system. If $A \xrightarrow{\phi}_a A'$ then $[\![A]\!] \xrightarrow{\phi}_a [\![A']\!] \mid A''$ for some A''.*

PROOF: By induction on the trace. A'' is composed of the system log and instances of the *Resolution* processes. □

PROOF OF THEOREM 3.7 (SKETCH) Let A be a high level system. Suppose that $A \xrightarrow{\phi}_i$ double-accepts. There exist configurations A_1, A_1', A_2, A_2', labels ϕ_1, ϕ_2, ϕ_3, ϕ_4, identifiers l, s, s' and principals $\mathcal{B}, \mathcal{M}, \mathcal{U}$ such that

(1) $A \xrightarrow{\phi_1}_a A_1$,

(2) $A_1 \xrightarrow{\phi_2}_a A_1'$ such that $\phi_2 = \mathsf{spend?}\,\mathcal{M}\,\mathcal{B}\,c\,l\,s\,\mathcal{U}$ or $\phi_2 = \mathsf{deposit?}\,\mathcal{B}\,\mathcal{M}\,l\,s\,\mathcal{U}$, or $\phi_2 = \_?\,(\,\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)\,)$,

(3) $A_1' \xrightarrow{\phi_3}_a A_2$,

(4) $A_2 \xrightarrow{\phi_4}_a A_2'$ such that $\phi_4 = \mathsf{spend?}\,\mathcal{M}\,\mathcal{B}\,c'\,l\,s'\,\mathcal{U}$ or $\phi_4 = \mathsf{deposit?}\,\mathcal{B}\,\mathcal{M}\,l\,s'\,\mathcal{U}$, or $\phi_4 = \_?\,(\,\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s')\,)$

The proof is by case analysis of the possible situations that could lead to double-accepting (different ϕ_2 and ϕ_4).

For instance, we suppose that $\phi_2 = \mathsf{spend?}\,\mathcal{M}\,\mathcal{B}\,c\,l\,s\,\mathcal{U}$ and $\phi_4 = \mathsf{spend?}\,\mathcal{M}\,\mathcal{B}\,c'\,l\,s'\,\mathcal{U}$. The LTS rule (Spend?) must have been applied, so A_1 must contain a process $\mathcal{M}[\mathsf{spend?}\,\mathcal{B}\;(x)\,(x_1)(x_2).P]$ in an evaluation context. Its implementation reduces to

$$\mathcal{M}[[\![P\{^{c}/_{x}\}\{^{\langle l,s\rangle}/_{x_1}\}\{^{\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)}/_{x_2}\}]\!] \mid \mathsf{repl}\;log\,!\,\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)]$$

which contains a replicated log record $\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)$. Similarly, the implementation of A_2 also reduces to a system that contains a replicated log record $\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s')$. We thus have

$$[\![A]\!] \xrightarrow{\phi_1\,;\,\phi_2\,;\,\phi_3\,;\,\phi_4}_a A' \mid \mathit{Resolution} \mid \mathsf{repl}\;log\,!\,\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s) \mid \mathsf{repl}\;log\,!\,\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s')$$

The *Resolution* process thus can consume one of each log entries and run the identify process

$$\mathsf{identify}\;\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s)\;\mathbf{rcp}(l,\mathcal{B},\mathcal{U},s')\;(x).bad\,!\,x$$

which reduces by rule IDENTIFY to $bad\,!\,\mathcal{U}$. So $[\![A]\!]$ detects cheating. □

Appendix D

Flexible pre- and post-conditions for F7

Lemma D.1 (Red Let Eval)**.** *If* $A \Downarrow M$ *and* $B\{M/x\} \Downarrow M'$ *then* ***let*** $x = A$ ***in*** $B \Downarrow M'$*.*

Proof: Suppose that $A \longrightarrow^* \nu\tilde{a}.A' \Rsh M$. By successive applications of the context rule (Red Let), we have **let** $x = A$ **in** $B \longrightarrow^*$ **let** $x = \nu\tilde{a}.A' \Rsh M$ **in** B. After renaming, by (Heat Res Let) and (Heat Fork Let), we have **let** $x = \nu\tilde{a}.A' \Rsh M$ **in** $B \Rrightarrow \nu\tilde{a}.$ **let** $x = A' \Rsh M$ **in** $B \Rrightarrow \nu\tilde{a}.A' \Rsh$ **let** $x = M$ **in** B. By (Red Let Val) we have **let** $x = M$ **in** $B \longrightarrow B\{M/x\}$. By hypothesis, $B\{M/x\} \longrightarrow^* \nu\tilde{b}.B' \Rsh M'$ for some $\tilde{b}, B'$. By (Red Res) and (Red Fork 2) $\nu\tilde{a}.A' \Rsh$ **let** $x = M$ **in** $B \longrightarrow^* \nu\tilde{a}.\nu\tilde{b}.A' \Rsh B' \Rsh M'$ so by transitivity and by applying the definition of $\Downarrow$, we have **let** $x = A$ **in** $B \Downarrow M'$. □

Lemma D.2 (Typ Seq Assume)**.** *If* $\Gamma \vdash e : T'$*,* $T' = (x : P)\{C\}$*, and* $fv(C') \subseteq dom(\Gamma)$*, then* $\Gamma \vdash$ ***assume*** C'*;* $e : (x : P)\{C \wedge C'\}$*.*

Proof: Since $\Gamma \vdash e : T'$, we have $\Gamma \vdash \diamond$. Since we also assume $fv(C') \subseteq dom(\Gamma)$, we can apply the rule (Typ Assume) yielding $\Gamma \vdash$ **assume** $C' : (\_ : unit)\{C'\}$ (F-Assume). From the hypothesis $\Gamma \vdash e : T'$ we obtain $\Gamma, C' \vdash e : T'$. Then by (Sub Refine Right) and (Sub Refl) we derive the subtyping judgement $\Gamma, C' \vdash (x : P)\{C\} <: (x : P)\{C \wedge C'\}$. Finally, we use the rule (Exp Subsum), yielding $\Gamma, C' \vdash e : (x : P)\{C \wedge C'\}$ (F-And). Then we apply the rule (Typ Let) to the formulas (F-Assume) and (F-And) yielding $\Gamma \vdash$ **assume** C'; $e : (x : P)\{C \wedge C'\}$. □

Proof of Lemma 6.1 Let A be a closed expression where *Call* and *Return* do not occur.

Evaluation (⇒) Suppose that for some value M we have $A \longrightarrow^* \nu\tilde{a}.e' \Rsh M$. We show that $[\![e]\!]_E \Downarrow [\![M]\!]_E$ by induction on the number n of reduction steps. If $n = 0$, we have $A = \nu\tilde{a}.e' \Rsh M$, and by definition of the translation we have $[\![A]\!]_E = \nu\tilde{a}.[\![e']\!]_E \Rsh [\![M]\!]_E$ and so $[\![e]\!]_E \Downarrow [\![M]\!]_E$.

The reductions of $[\![A]\!]_E$ are the same as the reductions of A, augmented with the forked assumptions introduced by the translation as follows. We perform a case analysis on the first used reduction rule.

- Suppose that the first used reduction rule is (Red Rec Fun). Since all functions are annotated we have $A = (M : T)\ N$ and

$$(M : T)\ N \longrightarrow e\{N/x\}\{M/f\} \Downarrow R$$

where $M =$ (**rec** f:T. **fun** $x \to e$). Then we have

$$\begin{aligned}[\![(M : T)\ N]\!]_E &\triangleq (\textbf{let } r = [\![M]\!]_E\ [\![N]\!]_E \textbf{ in assume } \mathit{Return}(f,x,r);\ r)\\ &\triangleq (\textbf{let } r = (\textbf{rec } f{:}T.\ \textbf{fun } x \to \textbf{assume } \mathit{Call}(f,x); [\![e]\!]_E)[\![N]\!]_E \textbf{ in}\\ &\qquad \textbf{assume } \mathit{Return}(f,x,r);\ r)\end{aligned}$$

First the expression let-bound to r reduces according to the rule (Red Rec Fun):

$$(\textbf{rec } f{:}T.\ \textbf{fun } x \to \textbf{assume } \mathit{Call}(f,x);[\![e]\!]_E)\ [\![N]\!]_E \longrightarrow (\textbf{assume } \mathit{Call}(f,x);[\![e]\!]_E)\{[\![N]\!]_E/x\}\{[\![M]\!]_E/f\}$$

followed by a series of (Red Let Eval) reductions. We have

(1) By (Heat Assume), $\textbf{assume } \mathit{Call}([\![M]\!]_E,[\![N]\!]_E) \longrightarrow \textbf{assume } \mathit{Call}([\![M]\!]_E,[\![N]\!]_E) \Rsh ()$, so $\textbf{assume } \mathit{Call}([\![M]\!]_E,[\![N]\!]_E) \Downarrow ()$.

(2) Since $e\{N/x\}\{M/f\} \Downarrow R$, by induction hypothesis we have $[\![e]\!]_E\{[\![N]\!]_E/x\}\{[\![M]\!]_E/f\} \Downarrow [\![R]\!]_E$.

(3) By applying (Red Let Eval) to (1) and (2), we get $[\![M]\!]_E\ [\![N]\!]_E \Downarrow [\![R]\!]_E$.

(4) By (Heat Assume), $\textbf{assume } \mathit{Return}([\![M]\!]_E,[\![N]\!]_E,[\![R]\!]_E) \Downarrow ()$.

(5) By applying (Red Let Eval) to (4) and $[\![R]\!]_E \Downarrow [\![R]\!]_E$, we get

$$(\textbf{assume } \mathit{Return}(f,x,r);\ r)\{[\![N]\!]_E/x\}\{[\![M]\!]_E/f\}\{[\![R]\!]_E/r\} \Downarrow [\![R]\!]_E$$

(6) By applying (Red Let Eval) to (3) and (5), we get

$$(\textbf{let } r = [\![M]\!]_E\ [\![N]\!]_E \textbf{in assume } \mathit{Return}(f,x,r);\ r) \Downarrow [\![R]\!]_E$$

– The proof uses simple induction arguments for other cases since their translation is homomorphic.

Evaluation ($\Leftarrow$) The proof is analogous to the proof of the ($\Rightarrow$) direction, by induction on the length of the derivation. The (Red Let Eval) reductions for the additional assumed formulas are omitted in the system before translation.

Safety ($\Rightarrow$) Suppose that A is safe, and for some A' and S, $A \to^* A'$ and $A' \Rrightarrow S$. By induction on the length of derivation and case analysis of the first reduction. (Base case) If $A' = A$ and $S = \nu\tilde{a}.(\prod \textbf{assume } C_i) \Rsh (\prod c_j!M_j) \Rsh \prod \mathcal{L}_k\{e_k\}$ then $[\![A]\!]_E \Rrightarrow \nu\tilde{a}.(\prod \textbf{assume } C_i) \Rsh (\prod c_j!M_j) \Rsh \prod [\![\mathcal{L}_k]\!]_E\{[\![e_k]\!]_E\}$. Since A is safe, if $e_k = \textbf{assert } C_k$ then $\{C_1 \dots C_m\} \vdash C_k$. But since $[\![\textbf{assert } C_k]\!]_E = \textbf{assert } C_k$, we have that $[\![A]\!]_E$ is also safe.

(Inductive case) We suppose that the lemma holds for all A' such that $A \to A'$, and we show that it holds also for A.

(Red Assert) If $A = \textbf{assert } C$ then $[\![A]\!]_E = A$ so A is safe iff $[\![A]\!]_E$ is safe.

(Red Rec Fun) If $A = (M : T)\ N$ where $M = (\textbf{rec } f{:}T.\ \textbf{fun } x \to e)$, then we have

$$(M : T)\ N \to e\{N/x\}\{M/f\}$$

$$\begin{aligned}[\![(M : T)\ N]\!]_E &\triangleq (\textbf{let } r = [\![M]\!]_E\ [\![N]\!]_E \textbf{in assume } \mathit{Return}(f,x,r);\ r) \\ &\triangleq (\textbf{let } r = (\textbf{rec } f{:}T.\ \textbf{fun } x \to \textbf{assume } \mathit{Call}(f,x);[\![e]\!]_E)[\![N]\!]_E \textbf{in assume } \mathit{Return}(f,x,r);\ r)\end{aligned}$$

First the expression let-bound to r reduces according to the rule (Red Rec Fun):

$$\begin{aligned}&(\textbf{rec } f{:}T.\ \textbf{fun } x \to \textbf{assume } \mathit{Call}(f,x);[\![e]\!]_E)\ [\![N]\!]_E \\ &\longrightarrow (\textbf{assume } \mathit{Call}(f,x);[\![e]\!]_E)\{[\![N]\!]_E/x\}\{[\![M]\!]_E/f\} \\ &\Rrightarrow \textbf{assume } \mathit{Call}([\![M]\!]_E, [\![N]\!]_E) \Rsh ([\![e]\!]_E\{[\![N]\!]_E/x\}\{[\![M]\!]_E/f\})\end{aligned}$$

By induction hypothesis, since $e\{N/x\}\{M/f\}$ safe, $[\![e]\!]_E\{[\![N]\!]_E/x\}\{[\![M]\!]_E/f\}$ is also safe; so with an additional **assume** it is also safe.

(Other rules) The translation of the expressions involved in other reduction rules is homomorphic, so the induction principle applies.

Safety (⇐) The proof is symmetric to Safety (⇒). By hypothesis A does not use *Call* or *Return*. So if $[\![A]\!]_E$ is safe, then the absence of assumes of *Call* and *Return* does not influence the asserts, so A is also safe.

Typing (⇒) We use the typing derivation of $E \vdash A : T_e$ to construct the typing derivation of $E \vdash [\![A]\!]_E : T_e$ (both in RCF).

- Typing derivations of non-functional values are the same for the translated values.
- If A is a function $A =$(**rec** *f*:T. **fun** $x \to e$), then $T_e = x : T_1 \to T_2$ and $[\![A]\!]_E =$(**rec** *f*:T. **fun** $x \to$ **assume** *Call*(*f*,*x*); $[\![e]\!]_E$)
  Let $E' = E, f : T, x : T_1$. The derivation for A is:
  $$\frac{E \vdash T_e <: T \qquad E' \vdash e : T_2 (H - Body)}{E \vdash A : T_e} (Fun)$$
  We construct the derivation for $[\![A]\!]_E$ using the induction hypothesis on (H-Body):
  $$\frac{E \vdash T_e <: T \qquad \dfrac{\dfrac{E' \vdash [\![e]\!]_E : T_2 \qquad (H - Body, induction)}{E', Call(f,x) \vdash [\![e]\!]_E : T_2} (Strengthen)}{E' \vdash \textbf{assume } Call(f,x); [\![e]\!]_E : T_2} (Assume, Let)}{E \vdash [\![A]\!]_E : T_e} (Fun)$$
- If A is a function application $A =(M{:}T)\ N$, then $T = x : T_1 \to T_2$ and $T_e = T_2\{N/x\}$) and
  $[\![A]\!]_E =$**let** $r = [\![M]\!]_E\ [\![N]\!]_E$ **in assume** *Return*(*f*:T,*x*,*r*); r)
  The typing derivation for A is:
  $$\frac{\dfrac{E \vdash M : T \qquad (H1)}{E \vdash (M : T) : T} (Annot) \quad E \vdash N : T_1 (H2)}{E \vdash (M : T)N : T_2\{N/x\}} (App)$$
  We construct the derivation for $[\![A]\!]_E$ using the induction hypotheses on (H1) and (H2):
  $$\frac{\dfrac{E \vdash [\![M]\!]_E : T\ (H1, ind) \qquad E \vdash [\![N]\!]_E : T_1\ (H2, ind)}{E \vdash [\![M]\!]_E\ [\![N]\!]_E : T_2\{N/x\}} (App) \quad \Delta}{E \vdash \textbf{let } r = [\![M]\!]_E\ [\![N]\!]_E \textbf{ in assume } Return(f{:}T,x,r); r : T_2\{N/x\}} (Let)$$
  where $\Delta = \dfrac{E, r : T_2\{N/x\}, Return(f,x,r) \vdash r : T_2\{N/x\} (Var)}{E, r : T_2\{N/x\} \vdash \textbf{assume } Return(f,x,r); r : T_2\{N/x\}} (Assume, Let)$
- For other expressions, the derivations are constructed recursively.

Typing (⇐) We use the typing derivation of $E \vdash [\![A]\!]_E : T_e$ to construct the typing derivation of $E \vdash A : T_e$. The construction is mainly symmetric to that of Typing(⇒). We simplify the derivations for functions and function applications, by removing the assumes and let bindings introduced by the translation. Since neither A nor T_e can mention *Call* and *Return*, these formulas are never used to derive formulas in the typing derivation of $[\![A]\!]_E$. We can erase them from the environment in typing derivations for A. □

Proof of Lemma 6.2 **(⇒)** Let e be an expression, Γ a typing environment, T a type. Suppose that $\Gamma \vdash [\![e]\!]_E : T$ in RCF. By induction on the structure of the expression e, we construct a typing derivation of $\Gamma \vdash e : T$ in RCF_E.

- If e is a function **rec** *f*: T_f. **fun** $x \to e_2$, then T is of the form $x : T_1 \to T_2$ and

$$[\![e]\!]_E \triangleq \textbf{rec } f{:}T_f.\ \textbf{fun } x \to \textbf{assume } Call(f,x); [\![e_2]\!]_E$$

  The rule (Typ Fun) must have been used for the typing $\Gamma \vdash [\![e]\!]_E : T$ in RCF; it has two hypotheses: $\Gamma \vdash x : T_1 \to T_2 <: T_f$ (H-Sub) and $\Gamma, f : T_f, x : T_1 \vdash e_2 : T_2$ (H-Assume-Body). The judgment (H-Assume-Body) results from the application of rules (Typ Let) and (Typ Assume), with the hypothesis

$$\Gamma, f : T_f, x : T_1, Call(f,x) \vdash [\![e_2]\!]_E : T_2 \text{(H-Body)}$$

By induction hypothesis, we have

$$\Gamma, f : T_f, x : T_1, \mathit{Call}(f,x) \vdash e_2 : T_2 \text{(H-Body-Ind)} \quad \text{in RCF}_E$$

Now we construct the typing judgement $\Gamma \vdash e : T$ in RCF_E. by applying the rule (Typ Fun PrePost) to the hypotheses(H-Sub) and (H-Body-Ind).

– If e is a function application $(M{:}T_f)\ N$, then T_f is of the form $x : T_1 \to T_2$ and

$$[\![e]\!]_E \triangleq \textbf{ let } r = [\![M]\!]_E\ [\![N]\!]_E \textbf{ in assume } \mathit{Return}(M{:}T_f,N,r);\ r$$

The rule (Typ Let) must have been used for the typing $\Gamma \vdash [\![e]\!]_E : T$ in RCF; it has three hypotheses: $\Gamma \vdash [\![M]\!]_E\ [\![N]\!]_E : T_2\{N/x\}$ (H-App), $r \notin T_2$ (H-Fv), and $\Gamma, r : T_2\{N/x\} \vdash \textbf{assume } \mathit{Return}(M{:}T_f,N,r); r : T$ (H-Return). We apply (Typ Seq Assume) to the judgment (H-Return), yielding $\Gamma, r : T_2\{N/x\} \vdash \textbf{assume } \mathit{Return}(M{:}T_f,N,r); r : (r : P)\{C \wedge \mathit{Return}(M{:}T_f,N,r)\} = T$ where $T_2\{N/x\} = (r : P)\{C\}$.
The judgment (H-App) involves the hypotheses $\Gamma \vdash [\![M]\!]_E : T_f$ and $\Gamma \vdash [\![N]\!]_E : T_1$. By induction hypothesis, we have $\Gamma \vdash (M) : T_f$ (H-Fun-Ind) and $\Gamma \vdash N : T_1$ (H-Arg-Ind) in RCF_E.
From (H-Fun-Ind) we get $\Gamma \vdash (M : T_f) : T_f$ (H-Fun). In RCF_E we can apply the rule (Typ App PrePost) to the hypotheses (H-Fun) and (H-Arg-Ind), and obtain $\Gamma \vdash ((M : T_f)\ N) : (r : P)\{C \wedge \mathit{Return}(M{:}T_f,N,r)\} = T$.

– otherwise, the translation is a homomorphism, and the induction hypotheses apply.

($\Leftarrow$) Let e be an expression, Γ a typing environment, T a type. Suppose that $\Gamma \vdash e : T$ in RCF_E. By induction on the structure of the expression e, we construct a type derivation of $\Gamma \vdash [\![e]\!]_E : T$ in RCF.

– If e is a function $\textbf{rec } f{:}\ T_f.\ \textbf{fun } x \to e_2$, then T is of the form $x : T_1 \to T_2$. Typing rule (Typ Fun PrePost) applies yielding hypotheses $\Gamma \vdash T <: T_f$ (H-Sub) and $\Gamma, f : T_f, x : T_1, \mathit{Call}(f,x) \vdash e_2 : T_2$ (H-Body). By induction hypothesis, we have $\Gamma, f : T_f, x : T_1, \mathit{Call}(f,x) \vdash [\![e_2]\!]_E : T_2$ (H-Body-Ind).
Now we construct an RCF typing judgment for $[\![e]\!]_E$.

$$[\![e]\!]_E = \textbf{rec } f{:}T_f.\ \textbf{fun } x \to \textbf{assume } \mathit{Call}(f,x); [\![e_2]\!]_E$$

From (H-Body-Ind) by applying (Typ Assume) and folding (Typ Let) we obtain $\Gamma, f : T_f, x : T_1 \vdash \textbf{assume } \mathit{Call}(f,x); e_2 : T_2$ (H-Body-Assume). We can fold the definition of $[\![e]\!]_E$ and by applying rule (Typ Fun) to the judgments (H-Body-Assume) and (H-Sub) we obtain $\Gamma \vdash [\![e]\!]_E : T$ in RCF.

– If e is a function application $M'\ N$, since we all functional values are annotated, we know that $M' = (M : T_f)$ for some T_f. Thus we have $e = (M{:}T_f)\ N$, and T_f is of the form $x : T_1 \to T_2$, and $T_2 = (r : P)\{C\}$. Typing rule (Typ App PrePost) applies yielding hypotheses $\Gamma \vdash (M{:}T_f) : T_f$ (from which we obtain $\Gamma \vdash M : T_f$ (H-Fun)) and $\Gamma \vdash N : T_1$ (H-Arg) with $T = (r : P)\{C \wedge \mathit{Return}(M{:}T_f,N,r)\}\{N/x\}$. By induction hypothesis, we have $\Gamma \vdash [\![M]\!]_E : T_f$ (H-Fun-Ind) and $\Gamma \vdash [\![N]\!]_E : T_1$ (H-Arg-Ind).
Now we construct an RCF typing judgment for $[\![e]\!]_E$.

$$[\![e]\!]_E = \textbf{ let } r = [\![M]\!]_E\ [\![N]\!]_E \textbf{ in assume } \mathit{Return}(M{:}T_f,N,r);\ r$$

We can apply the rule (Typ App) to (H-Fun-Ind) and (H-Arg-Ind) and obtain $\Gamma \vdash [\![M]\!]_E\ [\![N]\!]_E : T_2\{N/x\}$ (H-App).
From (Typ Seq Assume) we obtain $\Gamma, r : T_2\{N/x\} \vdash \textbf{assume } \mathit{Return}(M{:}T_f,N,r); r : T$ (H-Return). By combining the hypothesis $r \notin T_2$ with (H-Return) and (H-App), we can apply the rule (Typ Let).

– otherwise, the translation is a homomorphism, and the induction hypotheses apply.

□

Lemma D.3 (Predicate abstraction)**.** *Let σ be the function that replaces every fact Pre(M,N) with the formula (#Pre(f,x))[M/f,N/x] and every fact Post(M,N,O) with the formula (#Pre(f, x,r))[M/f,N/x,O/r].*

Let E be an environment that binds a function variable f. Let $axioms(E)$ be the formulas of E where Pre or Post occur, and $E = E', axioms(E)$. If $E' \vdash axioms(E)\sigma$ then

(1) *for any formula C,if $E \vdash C$ then $(E \vdash C)\sigma$;*

(2) *for any types T and T',if $E \vdash T <: T'$ then $(E \vdash T <: T')\sigma$;*

(3) *for any expression e and type T,if $E \vdash e : T$ then $(E \vdash e <: T)\sigma$.*

Proof of Theorem 6.5 The proof transforms the type derivation of e_0 into that of e_1 using the Lemma on predicate abstraction for derivation, subtyping and typing judgments.

Let σ be the substitution replacing *Pre* and *Post* with #*Pre* and #*Post* respectively.

– We consider a sub-derivation of $E \vdash (h\ f : T_4) : T_4$ within e_0. By assumption we know that $E \vdash h : T$, so $E \vdash h\ f : T_3[f/g]$ is derivable, and we have $E \vdash T_3[f/g] <: T_4$.

– By assumption we know that H can be given the type T in an environment with no assumptions on predicates *Pre* or *Post*. By Lemma D.3 for typing, within e_1 the expression H_f can be given type $T\sigma$. Then the application $H_f\ f$ can be given type $T_3[f/g]\sigma$.
Since E contains the binding $f : T_f$, we have $axioms(E) = \phi_{f:T_f}$, which hold trivially after applying the substitution σ: after macro-expansion, both judgments $E \vdash \forall x_1.\ C_1 \Rightarrow \#Pre(f,x_1)$ and $E \vdash \forall x_1,x_2.\ C_1 \wedge \#Post(f,x_1,x_2) \Rightarrow C_2$ are tautologies. Lemma D.3 for subtyping applies so we have $(E \vdash T_3[f/g] <: T_4)\sigma$.

□

Appendix E

Sample code

E.1 Multi-party protocol: client code

```
let player_code me server fmove addr =
  let sk = getPrivateKey usage me in
  let vks = getPublicKey usage server in
  let inout = connect addr in
  send inout (msg2bytes (Hello(me)));

  (match bytes2msg (recv inout) with Start (n,players,ssig1) → (* Receive invitation*)
  let challenge = payload2bytes (Start_data(n,players)) in
  if rsa_verify_prin usage server vks challenge ssig1 then (* Auth check *)
  ( assert(GameC(server,players,n));
  let mem1 : prin → prin list → bool = mem in
  let test1 = mem1 me players in if test1 then (* Check if I am a registered player
      *)
    let move = fmove n in (* Instantiate my move for this game *)
    let bmove = move2bytes (me,n,move) in
    let hash = sha1 bmove in (* Compute my commitment *)
    assume (Commits(me,n,hash)); (* Accept the game rules *)
    let smove = payload2bytes (Commit_data(n,hash)) in
    let mysig = Principals.rsa_sign usage me sk smove in
    send inout (msg2bytes (Sealed(hash,mysig))); (* Sign *)
  (match bytes2msg (recv inout) with HashList (hashes,ssig2,sigs) → (* All committed *)
  let commitment = payload2bytes (CommitList_data(n,players,hashes)) in
  let t1 = rsa_verify_prin usage server vks commitment ssig2 in (* Auth *)
  if t1 then ( assert(CommitListC(server,n,hashes));
    let r1 = zip3 players hashes sigs in (* Link players and commitments *)
    if forall_hash3 verify_hash3 n r1 then (* Auth check *)
            ( assert(ValidHashes3(n,r1));
            send inout (msg2bytes (Move (move) )); (* Reveal my move *)

  (match bytes2msg (recv inout) with MoveList (moves) → (* All revealed *)
    let evl = zip4 players hashes moves sigs in (* Create evidence *)
    if exists me move evl then (* Check my presence *)
    if forall_move verify_move n evl then (* Check validity of others' moves *)
    if forall_hash verify_hash n evl then (* Check validity of others' commitments,
        again *)
    if winning_move move moves then (* Check that victory is merited *)
```

```
    let e = ssig2, evl in (* Assemble complete evidence *)
    auditWins server n me move e; (* Use this evidence for audit*)
    print_string ((Data.istr me)^": I am the winner.\n")

    else ()
    else print_string ((Data.istr me)^": I lost.\n")
  | _ -> failwith "invalid move-list message")
   ) else failwith "bad hashes")
    else failwith "bad signature for hash list"
  | _ -> failwith "invalid hash-list message")
    else failwith "I am not a registered player" )
    else failwith "bad game start signature"
  | _ -> failwith "invalid challenge message")
```

E.2 List Library Interface

```
assume
 (∀x, u. Mem(x,x::u)) ∧
 (∀x, y, u. Mem(x,u) ⇒ Mem(x,y::u)) ∧
 (∀x, u. Mem(x,u) ⇒ (∃y, v. u = y::v ∧ (x = y ∨ Mem(x,v))))

val mem: x:α → u:α list → r:bool{r=true ⇒ Mem(x,u)}

val find: f:(α → bool) →
    u:α list{(∀x. Mem(x,u) ⇒ Pre(f,x))} →
    r:α { Mem(r,u) }

val forall: t:(α → bool) →
    xs:α list {(∀x. Mem(x,xs) ⇒ Pre(t,x))} →
    b:bool {(b = true ⇒ (∀x. Mem(x,xs) ⇒ Post(t,[x],true))) }

val exists: t:(α → bool) →
    xs:α list {(∀x. Mem(x,xs) ⇒ Pre(t,x))} →
    b:bool {(b = true ⇒ (∃x. Mem(x,xs) ∧ Post(t,[x],true))) }

val iter: f:(α → unit) →
    l:α list {(∀x. Mem(x,l) ⇒ Pre(f,[x]))} →
    r:unit {∀x. Mem(x,l) ⇒ Post(f,[x],())}

assume
 (∀x, y, u, v. Mem2((x,y),(x::u,x::v)) ∧
 (∀x, y, u, v, x', y'. Mem2((x,y),(u,b)) ⇒ Mem2((x,y),(x'::u,y'::v))) ∧
 (∀x, y, u, v. Mem((x,y),(u,v)) ⇒ (∃y1,y2,t1,t2. l1=y1::t1 ∧ l2=y2::t2
  ∧ ((y1=x ∧ y2=y) ∨ Mem2((x,y),(t1,t2))))))

val map: f:(α → β ) →
    l:α list {(∀x. Mem(x,l) ⇒ Pre(f,[x]))} →
    r:β list {∀x,y. Mem2((x,y),(l,r)) ⇒ Post(f,[x],y)}

assume (∀f. Hereditary(f) ⇔
  ( ∀v,acc,h,t. Inv(f,v,acc,hd::tl) ⇒ (Pre(f,[acc;hd]) ∧ (∀r. Post(f,[acc;hd],r) ⇒ Inv(f,v,r,tl)))))

val fold : v: γ → f:(α → β → α ) {Hereditary(f)} →
    acc:α →
    xs:β list {Inv(f,v,acc,xs)} →
    r:α { (xs = [] ∧ r=acc) ∨ Inv(f,v,r,[]) }
```